

DISS. ETH Nr. 11899

Parametrik in der Produkt Datenmodellierung

ABHANDLUNG
Zur Erlangung des Titels
DOKTOR DER TECHNISCHEN WISSENSCHAFTEN

der
EIDGENÖSSISCHEN TECHNISCHEN HOCHSCHULE
ZÜRICH

vorgelegt von

Thomas Bühlmann
Dipl. Informatik-Ingenieur ETH
geboren am 18. Juli 1967
von Basel-Stadt (BS) und Schlierbach (LU)

Angenommen auf Antrag von:
Prof. Dr. M. Engeli, Referent
Prof. Dr. M. Norrie, Korreferentin

1996

Vorwort

Nach Abschluss meines Informatikstudiums mit den Vertiefungen in “Theoretischer Informatik” und der “Theorie der Programmiersprachen” fand ich Aufnahme in Professor Engelis Forschungsgruppe am Institut für Werkzeugmaschinen und Fertigung.

Damit wurde ich als Informatiker mit einer Situation konfrontiert, die nicht typischer für dieses Metier sein konnte. Weg aus dem Kreis der reinen Informatik, Einarbeitung in ein neues Gebiet, Beschäftigung mit Fragen und Problemen bei denen die Informatik zur Lösungsfindung dienen sollte. Einen speziellen Stellenwert nahm dabei die Mitarbeit an der Entwicklung von *EXPRESS*, einer Datenmodelliersprache im Rahmen eines internationalen Standardisierungsprogrammes (STEP) ein. Das Projekt befand sich zum Zeitpunkt meiner Arbeitsaufnahme in einer kritischen Lage, und so konnte ich gleich zu Beginn die Freuden und Leiden eines internationalen Mammutprojektes mit ungefähr 150 weltweit verteilt arbeitenden Experten erleben.

Diese Erfahrung mag als gute Voraussetzung für die Erarbeitung einer Dissertation gelten, denn auch eine solche Arbeit zeigt Züge eines grösseren Projektes. Ebenfalls wie in einem “echten” Projekt stehen nur wenige Namen auf der Titelseite, aber ein erfolgreicher Abschluss ist trotzdem nur als Resultat einer Gruppenarbeit möglich. Aus diesem Grund danke ich

- meinem Doktorvater und Referenten Prof. Dr. M. Engeli für die Betreuung und Leitung der Arbeit, der Uebernahme des Referates und den praktischen Tips und Gesprächen,
- Frau Prof. Dr. M. Norrie, die nicht nur das Korreferat führte, sondern mir auch viele wertvolle Anregungen in fachlichen Belangen und für die Gestaltung der Dissertation lieferte,
- meinen Kollegen am Institut, im speziellen den Herren F. J. Metzger, T. Schnider und H.U. Wiedmer, die mich jeder auf seine Art unterstützten,
- und meinem Kollegen H.P. Müller, der mich stets anleitete, einfache, allgemeine und beherrschbare Lösungen zu suchen.

Nebst den wissenschaftlichen Begleitern dürfen ganz speziell die Kollegen und Freunde aus dem Privatleben nicht vergessen werden. So lassen sich Probleme im Projekt oft nur durch deren Anteilnahme und Unterstützung überwinden. Aus dem Grund geht mein spezieller Dank an

- meinen Bruder, der immer da ist, wenn man ihn braucht,
- meinen Eltern, die mir von früh auf versuchten, Wege zu ebnen und Fragen zu beantworten,
- und last but not least Martin de Urgoiti, der beinahe fester an die Vollendung dieser Arbeit glaubte als ich.

Yet the biggest sacrifice was made by my fiancée. She accepted an additional year of separation so that my goal could be reached. That is why I dedicate this work to you, Catherine. Thanks!

Zürich, 18, Juli 1996

Thomas Bühlmann

Abstract

Nowadays manufacturing industry faces stiffer pressure to reduce production costs. At the same time, the requirements on the products, as well as the degree of specialization, increase continuously. As a consequence, manufacturers produce fewer components of the final product in house and, instead, get them from subcontractors.

Fast and cost effective handling of such orders is only possible, if the subcontractor can rely on efficient exchange of the necessary product data. At a later time, similar data may have to be fed back to the orderer to comply with the documentation requirements prescribed by law and quality standards, such as ISO 9000. These goals can only be achieved reliably, if industry makes use of international standards. ISO 10303 with its title “Standard for the Exchange of Product Model Data” was designed for this purpose.

Current modeling languages, including *EXPRESS* on which the standard ISO 10303 is based, reveal shortcomings with respect to the requirements of the production industry. Hence the main goal of this thesis was to address these shortcomings through proposed extensions to *EXPRESS*. They should contribute to greater semantic content in data models thereby improving product model standards as well as data exchange.

The first extension proposed, addresses requirements to support the modeling of part libraries. Such digital libraries consist of a large amount of information which can only be handled if a generic description thereof is supported. For this purpose, we introduce partially evaluated objects which will become fully evaluated upon their actual use in the target environment.

Secondly, the work investigates problems which originate from the inflexible structure of hierarchical type systems known from many object oriented systems. Especially for product data, there are often several ways to choose classifications. All of them are equally valid and valuable. A rigid type system serves perfectly as a primary classification. It is enriched with a way to specify multiple classifications, which form an orthogonal concept to the type system. This suggestion proves powerful and offers support in the research areas of object and schema evolution, as well as in problems of versioning.

Finally, the work underlines the findings in type theory, that data modeling comprises structure *and* behavior. In order to support both, data modeling languages have to provide appropriate means to specify functional characteristics. The extension not only offers statically defined methods, but also dynamically associated methods the existence of which can be documented in the data schema.

Since all new features are presented with a formal semantic definition, they can be shown to be unambiguous and free of contradictions. The benefit of the extensions is indeed a higher semantic content of data models and the ability to convey this information through the documentation.

Zusammenfassung

Die heutige Produktindustrie sieht sich einem zunehmenden Kostendruck ausgesetzt. Gleichzeitig steigen die Anforderungen an die Produkte und damit der Spezialisierungsgrad von Teileherstellern. Das bewirkt, dass Hersteller einen immer grösseren Teil eines Gesamtprodukts nicht mehr selbst herstellen sondern von Zulieferern produzieren lassen.

Eine rasche und damit kostengünstige Bearbeitung von Aufträgen und Entwicklungen ist nur dann möglich, wenn Produktdaten effizient zwischen den Beteiligten ausgetauscht werden können. Dies wiederum lässt sich am besten durch internationale Standards erreichen, ein Ziel, das mit dem ISO Standard 10303 “Standard for the Exchange of Product Model Data” angestrebt wird.

Heutige Datenmodelliersprachen zeigen ebenso wie die in ISO 10303 eingesetzte Modelliersprache *EXPRESS* Schwachstellen. Darum stellt diese Arbeit anhand von drei *EXPRESS*-Erweiterungen dar, wie mehr semantische Information in die Datenschemata, und damit schlussendlich in die Standards und den Datenaustausch einfließen kann.

Im ersten Problemkreis werden die Anforderungen von Teilebibliotheken untersucht. Dabei geht es darum, grosse Informationsmengen durch generische Objekte auszutauschen. Diese sind nur teilweise evaluiert und werden erst beim konkreten Gebrauch in einem Zielsystem vollständig bestimmt.

Die zweite Erweiterung richtet sich an die Probleme, die sich aus der starren Klassenhierarchie von objektorientierten Systemen ergeben. Solche Typsysteme erlauben eine Klassifizierung nach *einem* Kriterium. Speziell bei Produktdaten ist dieser Ansatz ungenügend, da oft verschiedene, voneinander unabhängige Klassifizierungsmerkmale unterstützt werden müssen. Der verfolgte Ansatz zeigt auch Verwendungsmöglichkeiten in den Gebieten der Objekt- und Schemaevolution sowie in der Versionenbildung.

Zum Schluss wird noch die These vertreten, dass Datenmodellierung mehr als eine reine Strukturbeschreibung ist. Aus diesem Grund muss ein Modelliermittel die Definition von funktionalen Eigenschaften unterstützen. Die vorgeschlagene Lösung liefert nebst statisch gebundenen Methoden auch die Möglichkeit, dynamisch definierte Methoden im Datenschema dokumentieren zu können.

Alle Erweiterungen sind mit Hilfe einer formalen Semantik beschrieben und zeigen, dass sie sich widerspruchsfrei und effizient implementieren lassen. Der Gewinn der vorgestellten Erweiterungen liegt, wie das auch in Beispielen aufge-

zeigt wird, in einem deutlich höheren semantischen Gehalt solcher Datenmodelle. Dies kommt den Erfordernissen der besseren Dokumentation von Produkten entgegen und erlaubt einen sichereren Datenaustausch.

Inhaltsverzeichnis

Vorwort	i
Abstract	iii
Zusammenfassung	v
Inhaltsverzeichnis	vii
Abbildungsverzeichnis	ix
1 Einleitung	1
1.1 Motivation	1
1.2 Ziel dieser Arbeit	2
1.3 Gliederung der Arbeit	3
2 Grundlagen	6
2.1 CAD Systeme, Entwicklung und Lage	6
2.2 ISO 10303	7
2.2.1 Die Sprache <i>EXPRESS</i>	10
2.2.2 Das Physical File	11
2.3 Anforderungen	12
2.3.1 Parametrik	13
2.3.2 Sekundäre Klassifikationen	16
2.3.3 Operationelle Eigenschaften von Daten	20
2.4 Vergleich von Datenmodelliermitteln	22
2.4.1 Entity Relationship	24
2.4.2 Rumbaugh	24
2.4.3 Coad/Yourdon	25
2.4.4 Booch	26
2.4.5 Das OM Datenmodell von Norrie	28
3 Parametrik	29
3.1 Lösungsvorschlag	29
3.2 Syntaktische Erweiterungen	32
3.3 Semantische Grundlagen	35
3.4 Ausführlicheres Beispiel	41
3.5 Zusammenfassung	42

4	Klassifizierungen nach mehreren Kriterien	44
4.1	Lösungsvorschlag	44
4.2	Syntaktische Erweiterungen	45
4.3	Semantische Grundlagen	46
4.4	Evolution	55
4.4.1	Objektevolution	56
4.4.2	Schemaevolution	57
4.5	Ausführlichere Beispiele	59
4.6	Zusammenfassung	63
5	Methoden	64
5.1	Syntaktische Erweiterungen	64
5.2	Semantische Grundlagen	67
5.3	Ausführlichere Beispiele	69
5.4	Zusammenfassung	73
6	Ausblick	74
6.1	Erreichte Ziele	74
6.2	Grenzen der Erweiterungen	75
6.3	Zukünftige Schritte	76
	Glossar	77
	Literaturverzeichnis	80

Abbildungsverzeichnis

1.1	Modellierung eines Kegelstumpfes	1
1.2	Modellierung eines Kegelstumpfes, Variante	2
1.3	Kapitelstruktur	4
2.1	Effekt der direkten Formatübersetzung	7
2.2	Effekt eines neutralen Datenformats	8
2.3	Aufbau von STEP anhand der Standarddokumente	9
2.4	<i>EXPRESS</i> geometrische Grundkörper	10
2.5	Physical File: geometrische Grundkörper	12
2.6	Ausprägung einer Schraubenfeder nach math. Modell	14
2.7	Drei Kennlinien einer Schraubenfeder	15
2.8	parametrisch definierte Schraube	15
2.9	SQL-Modell, Dokumentverwaltung	18
2.10	SQL Modell, techn. Zeichnung	19
2.11	<i>EXPRESS</i> -Modell, Dokumentverwaltung	20
2.12	Modell, Zeichnungsverwaltung	21
2.13	Entwicklungsstufen für eine Anwendung	23
2.14	Modelle der objektorientierten Entwicklung	27
3.1	konzeptueller Austausch parametrischer Daten	30
3.2	Schraubenfeder, Datenmodell und Austauschfile	31
3.3	Schraube, Datenmodell und Austauschfile	32
3.4	Datendefinitionssprache/Austauschmodell alt	34
3.5	Datendefinitionssprache/Austauschmodell neu	35
3.6	Schraube, Bild und Modell	41
3.7	Flansch, Bild und Modell	42
3.8	Austauschfile für Flansch und Schraube	43
4.1	Modell, Zeichnungsverwaltung	46
4.2	Vererbung von Zuständen	48
4.3	Wertevergleichsalgorithmus	55
4.4	Teilemodell, <i>EXPRESS-G</i>	59
4.5	Teilemodell, <i>EXPRESS-Text</i>	60
4.6	Flansch, <i>EXPRESS-G</i>	61
4.7	Flansch, <i>EXPRESS-Text</i>	62
4.8	Kodefragment, Flansch mit Versionen	62
5.1	Beispiel für Methoden in <i>EXPRESS</i>	66
5.2	Kreuzassoziation von Methoden in C++	66
5.3	Teilemodell mit Methoden, <i>EXPRESS-Text</i>	70
5.4	Flansch mit Methoden, <i>EXPRESS-Text</i>	71
5.5	Verwendung von Methoden, <i>EXPRESS-Text</i>	71

5.6	dynamisch bindbare Methode	72
5.7	Austauschfile mit dyn. Methode	73

Kapitel 1

Einleitung

1.1 Motivation

Ausgangspunkt für die vorliegende Arbeit war die Erkenntnis, dass viele Probleme der Produktdatenmodellierung noch nicht gelöst sind. Obschon mit der Verwendung des Entity-Relationship Konzepts viele Datenmodelle entwickelt und dokumentiert werden können, lässt sich dieser Ansatz nur schwer auf die Modellierung von Produktdaten anwenden.

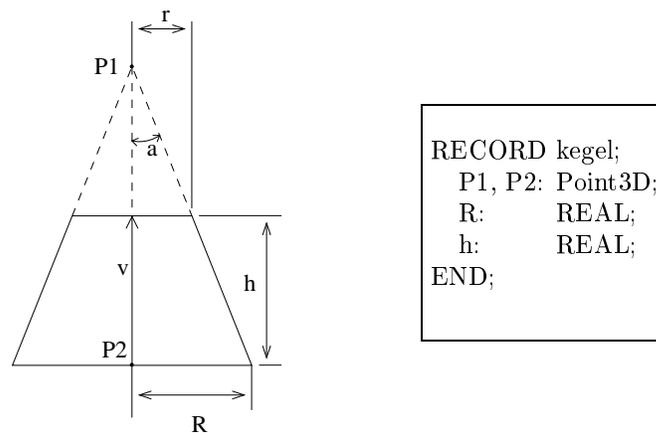


Abbildung 1.1: Modellierung eines Kegelstumpfes

Produktdaten zeichnen sich dadurch aus, dass sie eine starke Vernetzung durch Beziehungen zwischen einzelnen Teilen aufweisen. Der Versuch, Produkte von der Entwicklung bis zur Entsorgung durch Dateninstanzen beschreiben zu können, zusammen mit der immer längeren Archivierungspflicht von Daten, resultiert ausserdem in der Notwendigkeit, dass sich die Modelle und Instanzen entwickeln können. Im Gegensatz zu Anwendungen mit kurzlebigeren Informationen können die Datenbestände bei einem Wechsel des Datenschemas jeweils nicht reorganisiert werden.

Zur Verdeutlichung der Anforderungen denke man sich ein Datenmodell eines im Raum orientierten Kegelstumpfes (siehe Abb. 1.1). Schon für dieses einfache Beispiel gibt es je nach Wahl der Parameter verschiedene Möglichkeiten der Modellierung. Ein oft verwendeter Ansatz ist in der Abbildung 1.1 in einer programmiersprachenähnlichen Syntax dargestellt. Dieser ist solange gut, als die künstliche Singularität beim Uebergang vom Kegelstumpf in einen Zylinder nicht auftritt. In diesem Fall, der in der Praxis zum Beispiel in der Druckindustrie vorkommt, eignet sich eine Parametrisierung mit Hilfe des Oeffnungswinkels besser (siehe Abb. 1.2).

```

RECORD kegel;
  P2: Point3D;
  a:  REAL;
  R:  REAL;
  v:  Vector3D;
END;

```

Abbildung 1.2: Modellierung eines Kegelstumpfes, Variante

Das Datenmodell kann zwar bei Bedarf der Situation angepasst werden, doch stellt sich bei jeder Anpassung die Frage nach der Konvertierung der alten Daten, respektive nach der Interoperabilität von geometrischen Objekten zwischen dem ersten und dem zweiten Modell. Optimal wäre ein *dynamisch getyptes* System oder ein einziger, und damit trivialerweise kompatibler Datentyp, der beide Modellierungsvarianten unterstützt.

Nebst diesem Problem hat der in den letzten Jahren stetig gestiegene Kostendruck dazu geführt, dass der Aufwand und vor allem die Vorgehensweise in der Entwicklung die wesentlichsten Beiträge an die Endkosten liefern. Zwar werden nur etwa 10% der direkten Kosten durch die Entwicklung verschlungen [Hed90], doch bestimmt die Entwicklung die Lieferzeit und sämtliche nachfolgenden Kosten massgeblich [LS93]. Da zudem "durchschnittlich 80% der Konstruktionslösungen eines Betriebs komplett oder nur mit geringen Aenderungen wiederverwendet werden können" [Hed90], ist der Schritt zu einem Datenmodell, das *parametrisch definierte Instanzen* ermöglicht zwingend [LS93].

1.2 Ziel dieser Arbeit

Mit den im letzten Unterkapitel aufgeführten Problemen sah sich eine internationale Kommission der ISO (International Standardization Organization) konfrontiert. Das Ziel der Gruppe bestand darin, Produktdatenmodelle für verschiedene Industriezweige zu standardisieren. Obwohl der daraus entstandene

Standard [ISO94a] viele Probleme im Datenaustausch lösen konnte, erfüllen die verwendeten Datenmodelliermittel nicht sämtliche Wünsche.

Daher ist das Ziel dieser Arbeit, die in [ISO94b] standardisierte Datenmodelliersprache *EXPRESS* [ISO94b] um Möglichkeiten für eine detailliertere Spezifikation von Datentypen zu erweitern. Darunter fallen vor allem

- die Möglichkeit der Spezifikation operationeller Eigenschaften von Datentypen,
- die Unterstützung der dynamischen Kombinierbarkeit von Instanzen, damit Instanzen zu verschiedenen Zeiten unterschiedliche Eigenschaften annehmen können,
- und schlussendlich die Definition und Handhabung von parametrisch definierten Instanzen.

Zu diesen Zielen ist grundsätzlich zu bemerken, dass ein Datenmodell, besonders wenn es für die Standardisierung eingesetzt wird, nur brauchbar ist, wenn das verwendete Spezifikationsmittel auf präzisen Definitionen basiert. Bei der ersten Version von *EXPRESS* [ISO94b] ist die formale Spezifikation der Semantik unterblieben und soll aus diesem Grund für die in dieser Arbeit vorgeschlagenen Erweiterungen eingeführt werden.

Ausserdem kann nicht oft genug wiederholt werden, dass ein Datenmodell keinem Selbstzweck dient, sondern als Grundlage für eine Speicherung respektive den Datenaustausch gilt. Aus noch darzulegenden Gründen ist die Verwendung eines neutralen Formates zur Speicherung von Instanzen zweckmässig. Eine solche Formatdefinition existiert nebst *EXPRESS* als eigener Standard [ISO94c]. Damit wird auch klar, dass die gesetzten Ziele nur zu erreichen sind, wenn dieses Format ebenfalls angepasst wird.

Schlussendlich ist zu erwähnen, dass es noch keine allgemein akzeptierte Theorie zu objektorientierten Datenbanken gibt. Derartige Systeme sind noch einem starken Wandel und verschiedenen Experimenten unterworfen. Aus diesem Grund ist es nicht verwunderlich, dass die hier untersuchten Eigenschaften höchstens ansatzweise in Systemen realisiert sind. Andererseits zeigt aber die Zahl der Veröffentlichungen auf dem Gebiet der objektorientierten Modellierung, dass die Forschung ein reges Interesse an Verbesserungen auf diesem Gebiet zeigt.

1.3 Gliederung der Arbeit

Da diese Arbeit auf einer speziellen Architektur mit einem eigenen Konzept der Datenbeschreibung und des Informationsaustausches beruht, werden im nächsten Kapitel die Grundlagen für das Verständnis der folgenden Teile vermittelt. Soweit als notwendig wird dabei auch auf die speziellen Anforderungen, wie sie aus der Standardisierung resultieren, eingegangen. Ebenfalls in diesem Kapitel wird die Definitionssprache *EXPRESS* mit dem Entity Relationship Modell

und objektorientierten Datenmodellen bezüglich ihrer Modelliereigenschaften verglichen (s. Kapitelstruktur in Abb. 1.3).

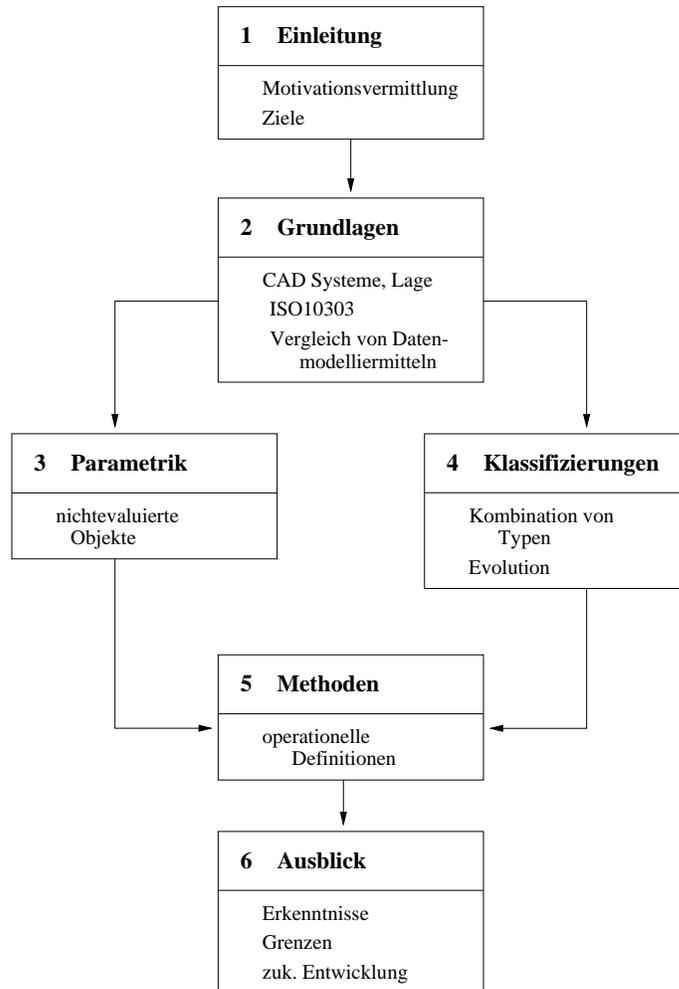


Abbildung 1.3: Kapitelstruktur

Kapitel 3 behandelt den Problemkreis der Parametrik. Im Rahmen dieser Arbeit wird darunter die Definition und der Austausch von teilweise evaluierten Objekten verstanden. Den Anforderungen der aufkommenden parametrischen CAD Systemen lässt sich Rechnung tragen, indem man *EXPRESS* und das Austauschformat derart erweitert, dass algorithmische Ausdrücke referenzierbar werden. Das Kapitel ist in die drei Teile “syntaktische Erweiterung”, “semantische Spezifikation” und Anwendung anhand einiger Beispiele gegliedert.

Die Frage der Kombinierbarkeit verschiedener Datentypen lässt sich unabhängig von der Parametrik als eigener Problemkreis behandeln. In Kapitel 4 wird daher eine Lösung gesucht, wie sich dynamische Aspekte, die in der Forschung auch unter dem Begriff “Rollen” behandelt werden, in *EXPRESS* inte-

grieren lassen. Wesentlich ist dabei, dass man Klassenhierarchien als Ausdruck einer Primärklassifizierung betrachtet, zusätzlich aber noch Sekundärkriterien einführt. Ein solcher Ansatz, ebenfalls mit einem syntaktisch und semantisch orientierten Unterkapitel, führt schliesslich auf die Frage der Daten- und Schemaevolution. Diesem Thema ist ein eigenes Unterkapitel gewidmet. Schlussendlich wird die Anwendbarkeit der Erweiterungen wiederum an Beispielen aufgezeigt.

Interessanterweise führen die beiden gut trennbaren Kapitel zu einer Schlussfolgerung, die sich in der Typtheorie inzwischen etabliert hat. So lassen sich Datentypen nur als untrennbare Kombination von Struktur und Operationen definieren. Aus diesem Grund wird im letzten technischen Kapitel eine Erweiterung von *EXPRESS* um Methoden vorgestellt. Die im Kapitel über die Parametrik gemachten Vorschläge erlauben dazu eine recht weitgehende Definition. So können sowohl statisch wie auch dynamisch definierte Methoden dokumentiert werden. Dies führt zu flexiblen Möglichkeiten in der Datenmodellierung, was sich besonders für die Standardisierung als Vorteil erweist.

Die Erkenntnisse aus dieser Arbeit, sowie mögliche zukünftige Schritte, soweit sie sich aus den hier untersuchten Anforderungen ergeben, bilden den Inhalt des letzten Kapitels.

Kapitel 2

Grundlagen

2.1 CAD Systeme, Entwicklung und Lage

Die Entwicklung der CAD Systeme verläuft in vielerlei Hinsichten ähnlich wie bei den Datenbanken. So wurden in einem ersten Schritt die mechanischen Zeichenbretter durch Softwarelösungen ersetzt. Gleich wie bei den Datenbanken handelte es sich dabei um Insellösungen die auf proprietären Datenmodellen beruhten. Als Folge davon konnten elektronisch gespeicherte Daten kaum auf andere Systeme übertragen werden.

Im einem nächsten Schritt fand eine Spezialisierung der CAD Systeme statt. Gleichzeitig erfolgte der Wunsch nach einem durchgängigeren Fluss der Daten. Um die Wiederverwendung der erarbeiteten Informationen zu gewährleisten, hatten Zulieferer entweder dasselbe System anzuschaffen wie der Auftraggeber, oder eine Datenkonvertierung musste mit Hilfe von sogenannten Prozessoren (“processor”) erfolgen. Unter Prozessoren werden in diesem Zusammenhang Softwareprogramme verstanden, die eine Abbildung von Daten zwischen den internen Datenmodellen zweier CAD Systeme durchführen. In STEP bezeichnet man die Konverter in das neutrale Austauschformat als Präprozessoren und diejenigen vom Austauschformat ins CAD-interne Datenformat als Postprozessoren. Bei solchen Umwandlungen treten allerdings zwei Probleme auf. Zum einen können die Prozessoren oft nicht alle Eigenschaften eines Datenmodells abbilden (*Informationsverlust!*). Das führt dann zu einer teuren und fehleranfälligen Nachbearbeitung der übermittelten Daten. Das zweite Problem liegt in der quadratischen Zunahme der notwendigen Prozessoren, um den Austausch zwischen allen System zu ermöglichen (siehe Abb. 2.1).

Solchen Situationen kann nur durch Normen oder Standards abgeholfen werden. Derartige Abmachungen können sowohl in Branchen als auch auf nationaler oder internationaler Ebene entwickelt und in Kraft gesetzt werden. In den Achtzigerjahren entstanden mehrere nationale Normen für den Datenaustausch zwischen Geometriesystemen. Die bekanntesten und teilweise auch heute noch verwendeten sind IGES (**I**nitial **G**raphics **E**xchange **S**pecification [SW86, Smi91]), SET (**S**tandard d’**E**change et de **T**ransfert [SET89]), VDA-FS (**V**erein der **A**utomobilindustrie – **F**lächen-**S**chnittstelle [VDA86, VDA87]) und VDA-IS (**V**DA IGES-**S**ubset [VDA89]). Sie alle zeichnen sich dadurch aus, dass für den

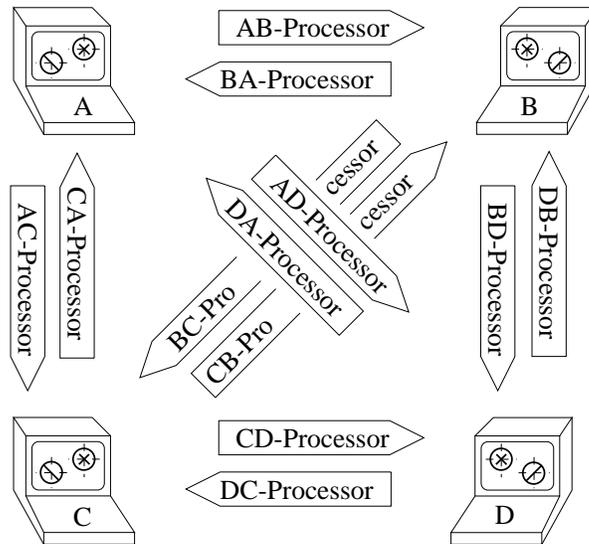


Abbildung 2.1: Effekt der direkten Formatübersetzung

Datenaustausch ein neutrales Format verwendet wird, und dass eine fixe Zahl von geometrischen Grundelementen existiert.

Mit einem neutralen Austauschformat reduziert sich die Zahl der notwendigen Prozessoren auf einen linearen Wert in Abhängigkeit der unterstützten Systeme (siehe Abb. 2.2). Allerdings zeigten diese Verfahren neue Probleme. So bestanden zu dieser Zeit sämtliche Standards aus natürlichsprachigen Beschreibungen. Datenmodelle auf dieser Basis führten zu vielerlei Interpretationen. Daher mussten Richtigstellungen respektive Interpretationsrichtlinien [IGE91] publiziert werden, damit die Implementierungen infolge verschiedener Interpretationen des Standards nicht wieder Inkompatibilitäten zwischen den Systemen erzeugten.

Als weiterer schwerwiegender Fehler erwies sich die fehlende Trennung von Austauschformat und Datenmodell. Technische Verbesserungen konnten nur schwierig in den Standard eingearbeitet werden, da jede Anpassung am Datenmodell neuer Definitionen für das Austauschformat bedurfte. Berücksichtigt man zudem die relativ lange Bearbeitungszeit, bis ein Standard offiziell angepasst ist, so wird klar, dass die Definitionen immer hinter dem Stand der Technik herlaufen mussten.

2.2 ISO 10303

Ein Treffen von Spezialisten im Jahre 1984 markiert den Beginn eines neuen Anlaufes in der Standardisierung des Produktdatenaustausches. Die relativ kleine Gruppe setzte sich zum Ziel, in kurzer Zeit einen neuen internationalen Standard für den Produktdatenaustausch unter dem Titel “**S**Tandard for the

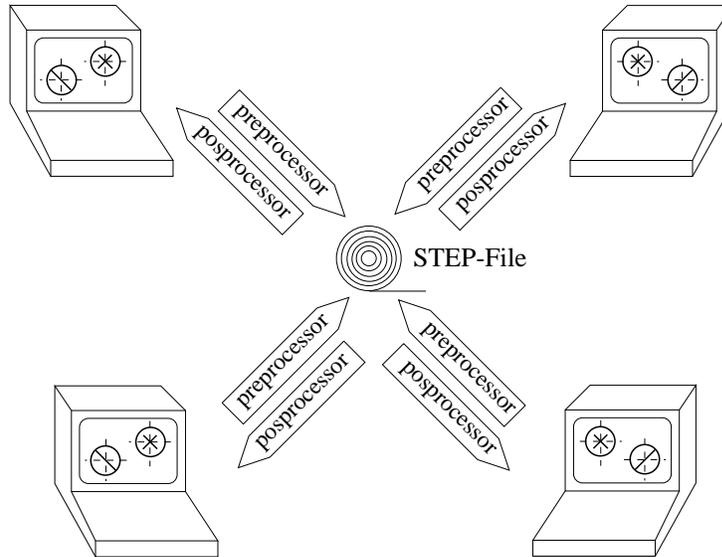


Abbildung 2.2: Effekt eines neutralen Datenformats

Exchange of Product Model Data” oder kurz STEP zu entwickeln.

Bedenkt man, dass bis anhin beinahe ausschliesslich Geometriedaten zwischen Systemen ausgetauscht werden konnten, so ist die Ambition oder Kühnheit aus dem Ziel ersichtlich, beschreibende Daten irgendwelcher Produkte zu erfassen, zu speichern und zwischen beliebigen CAD Systemen auszutauschen.

Mit dem Umfang und der Komplexität des Projektes wurde vollkommenes Neuland in der Standardisierung betreten. Aus dem Grund versuchten die Spezialisten, mit Hilfe einer organisatorischen Aufteilungen und einer klar umrissenen Architektur den Standardaufbau zu gliedern.

In Abbildung 2.3 ist der Aufbau der STEP-Architektur vereinfacht dargestellt. Als Grundlagen gelten die fundamentalen Prinzipien (ISO 10303-1), die in englischem Prosatext festgehalten sind. Das Dokument enthält allgemeine Erklärungen, Terminologiedefinitionen sowie Entwurfsprinzipien und -vorgehen für die Definition von Datenmodellen.

Die nächsten beiden Dokumente, die auch ausserhalb der Standardisierungsaktivitäten in STEP Anwendung finden, definieren die Datenmodelliersprache *EXPRESS* [ISO94b] und ein neutrales Austauschformat (ISO 10303-21 [ISO94c], “Physical File”). Mit diesen beiden Hilfsmitteln kann die in ISO 10303-1 festgehaltene Forderung nach einer formalen Definitionssprache für die Datenmodelle und einem allgemeinen Austauschkonzept für jegliche in *EXPRESS* definierten Datenmodelle erfüllt werden. Zusätzlich vereinfacht die Wahl einer formalen Modelliersprache die Kommunikation zwischen den internationalen Experten. Ein weiterer Vorteil dieses Ansatzes liegt in der Möglichkeit mit Unterstützung von Computerprogrammen die Konsistenz innerhalb und zwischen Datenmodellen

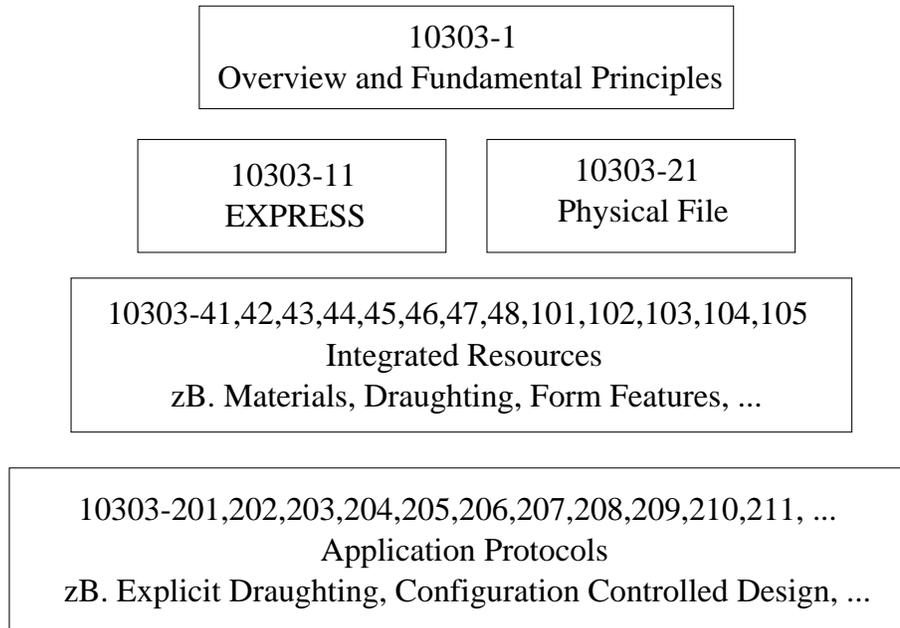


Abbildung 2.3: Aufbau von STEP anhand der Standarddokumente

sicherzustellen, was die Qualität dieser umfangreichen Standardserie beträchtlich verbessert.

Die nächste Dokumentgruppe besteht aus den integrierten Ressourcen. Sie definieren eine Art Klassenbibliothek, auf denen die branchenspezifischen Datenmodelle (“Application Protocol”) aufbauen. Ziel der Ressourcen war nebst einer Reduktion der Zahl von Datentypen oder Funktionen mit gleichem Namen, aber in verschiedenen Datenmodellen, dass die Verwendung gemeinsamer Grundklassen die Interoperabilität von Daten zwischen verschiedenen Branchen fördert.

Die letzte Gruppe besteht aus den bereits erwähnten Anwendungsprotokollen (“application protocol”). Dabei handelt es sich um die funktions- oder branchenspezifischen Datenmodelle. Diese Dokumente bauen auf den integrierten Ressourcen auf und stellen denjenigen Teil dar, der die Struktur und die Semantik von ausgetauschten Daten letztendlich standardisiert.

Mit der Entwicklung des Standards haben sich die technischen Möglichkeiten, Anforderungen und Lösungsansätze immer wieder verändert. Zugleich haben demokratische Strukturen in der internationalen Standardisierung die Verzögerung der Arbeiten gefördert. So erschien im Jahre 1994, zehn Jahre nach der Inangriffnahme des Projekts, eine erste Serie von standardisierten Dokumenten unter der Bezeichnung ISO-10303:1994E.

2.2.1 Die Sprache EXPRESS

Die Datenmodelliersprache *EXPRESS* ist das Resultat einer Synthese aus existierenden Programmiersprachen und Elementen aus SQL (Structured Query Language [DD93]). Dementsprechend sind die textuellen Definitionen recht gut lesbar, wie das aus Abbildung 2.4 anhand eines Beispiels, bestehend aus einer Kugel und einem geometrischen Punkt, ersichtlich ist.

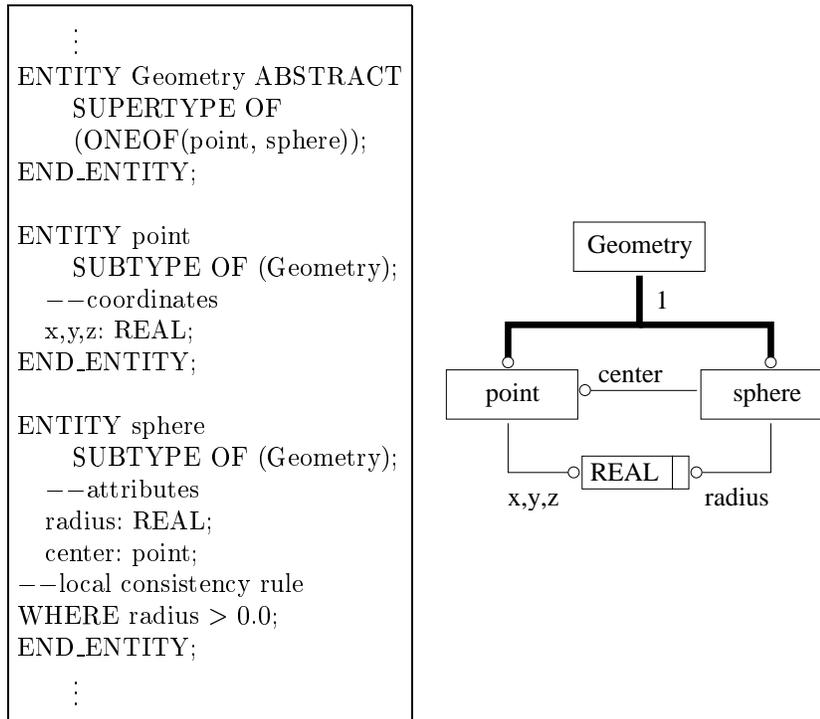


Abbildung 2.4: *EXPRESS* geometrische Grundkörper

In der gleichen Abbildung findet sich eine nach [ISO94b] standardisierte Darstellung (*EXPRESS-G*) desselben Modells. Darin erkennt man die Klassenhierarchie, gegeben durch die fett gezogenen Linien. Ebenfalls ersichtlich sind die mit feineren Linien gekennzeichneten Attribute.

Speziell an der Datenmodelliersprache sind die Konsistenzbedingungen, die allerdings nur aus der textuellen Repräsentation des Modells hervorgehen. In diesem Beispiel sind dies

- das Schlüsselwort “ABSTRACT”, das die Instanziierung eines Objekts vom Typ “Geometrie” verbietet.
- die Forderungen, dass sich die Typen “Point” und “Sphere” gegenseitig ausschließen (“ONEOF”). Dieses Modelliermittel kann verwendet werden um parallel zu den eigentlichen Klassenhierarchien eine statische Dokumentation von Rollendefinitionen zu spezifizieren.

- die Bedingung, dass der Kugelradius echt grösser als null ist (“local rule”),

Zu Beginn der Standardisierungsarbeit in STEP wurden die Anforderungen an ein Modellierhilfsmittel zusammengetragen [Wei86]. Da damals keine Datenmodelliersprache verfügbar war, die den Anforderungen genügte, wurde *EXPRESS* mit den folgenden Eigenschaften entwickelt:

- *EXPRESS* ist eine formal definierte Sprache. Daher können auf einfache Art Softwarewerkzeuge zur Verifizierung von Datenmodellen erzeugt werden.
- *EXPRESS* ist eine objektorientierte, hybride Sprache. Damit gibt es sowohl klassenähnliche Datentypen (“entity”) als auch Grunddatentypen (“simple data types”).
- *EXPRESS* kennt ein eigenes Vererbungsprinzip. Es sieht vor, dass Entitäten gleichzeitig mehreren Typen angehören können. Aus diesem Grund gibt es sprachliche Hilfsmittel, um die im Datenmodell erlaubten Typkombinationen zu spezifizieren.
- *EXPRESS* unterstützt den modularen Aufbau von komplexen Datenmodellen und stellt dazu ein Kapselungsprinzip zu Verfügung.
- *EXPRESS* kennt abgeleitete Attribute (“derived attributes”), um Informationen, die aus Attributen herleitbar sind, zu modellieren.
- *EXPRESS* erlaubt die Modellierung von Konsistenzbedingungen. So kann nebst der Struktur eines Datentyps oder Datenmodells auch ein Teil der Semantik vermittelt werden.
- *EXPRESS* kennt algorithmische Spezifikationen, um kompliziertere, abgeleitete Attribute und Konsistenzbedingungen spezifizieren zu können.
- *EXPRESS* kennt mit *EXPRESS-G* [ISO94b] eine graphische Darstellungsform, die sich sehr gut zur Entwicklung und Präsentation von Datenmodellen eignet.

Eine tiefere Einführung in die Eigenschaften dieser Modelliersprache lässt sich in [ISO94b, SW94, Owe93] finden.

2.2.2 Das Physical File

Für den Austausch von Instanzen müssen diese nach den Vorschriften in [ISO94c] kodiert werden. Dieser Standard trägt die volle Bezeichnung “Clear Text Encoding of the Exchange Structure” und erlaubt die Konstruktion des Austauschformates mit der Kenntnis des Datenmodells. Dabei erfolgt die Kodierung der evaluierten Objekte, die in STEP mit Instanzen bezeichnet werden, in ASCII [ASC86] und ist damit im Prinzip für den Menschen lesbar. Als Beispiel ist dies für das zuvor eingeführte Modell einer Sphäre in Abbildung 2.5 aufgezeigt.

Dieses Format eignet sich sehr gut für eine langfristige Archivierung oder den Datenaustausch auf magnetischen Bändern. Für den direkten Zugriff auf verteilte Datenbanken ist dieses System hingegen nicht geeignet. Aus diesem Grund sind zur Zeit verschiedene Zugriffsschnittstellen auf Datenbanken im Entstehen begriffen. Erwähnenswert ist eine allgemeine Zugriffsschnittstelle [Pri94], die sich in einem fortgeschrittenen Stadium befindet. Daneben haben auch Arbeiten zur Unterstützung der an Bedeutung gewinnenden Industriennorm CORBA (Common Object Request Broker Architecture) mit seiner Interface Definition Language (IDL [C⁺94]) begonnen.

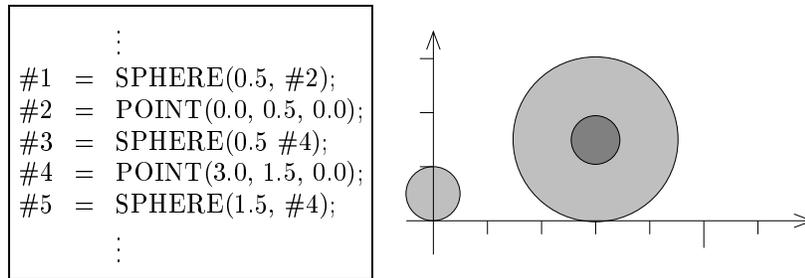


Abbildung 2.5: Physical File: geometrische Grundkörper

Da *EXPRESS* nie auf einer formalen Beschreibung der Semantik basierte, haben sich in der letzten Zeit Mängel in diesem Bereich gezeigt. Die fehlenden Definitionen zur Semantik sind meist aus der Kenntnis um das Austauschformat herleitbar, doch erzeugt das eine ungewollte Abhängigkeit der Teile und macht zudem die Erarbeitung neuer Austauschformate schwieriger als erwartet. Als weiteres Problem stellt sich die Tatsache, dass nicht mehr klar ist, welches Standarddokument nun zur Interpretation der Semantik herangezogen werden soll, respektive, dass sich die Definitionen widersprechen.

Dennoch muss man anerkennen, dass die wichtigsten Ziele durch die Entkoppelung von Austauschformat und Datenmodell mit der Publikation als internationaler Standard erreicht worden sind.

2.3 Anforderungen

Die folgenden Unterkapitel gehen auf die Anforderungen an eine zukünftige Datenmodelliersprache ein. Dabei darf man nicht vergessen, dass Anforderungen aus Anwendungsgebieten resultieren. Die hier untersuchten Erweiterungen sind aus diesem Grund eng mit der Entwicklung im CAD und CAM Bereich verknüpft und mögen in anderen Gebieten nur beschränkt gelten.

2.3.1 Parametrik

Unter den Herstellern von heutigen CAD Systemen werden parametrische Fähigkeiten als notwendig und immer wichtiger eingestuft [LS93]. Obwohl Parametrik in CAD Systemen nicht neu ist [EH74], hat sich im Bereich der Produktdaten bis heute keine allgemein akzeptierte Definition etablieren können. Unbestritten ist, dass parametrische Modelle noch Freiheitsgrade aufweisen und damit zur generischen Beschreibung von Produkten dienen können. Eine schlussendliche Spezifikation, also die Wandlung zu einer voll evaluierten Instanz, erfolgt erst zu einem späteren Zeitpunkt, häufig in einer anderen Umgebung, oft auch in einem anderen Informationssystem.

Meist werden Produkte in einem ersten Anlauf nicht parametrisch entwickelt. In der Praxis haben sich daher die folgenden, typischen Entstehungsszenarien für parametrische Definitionen ergeben:

- Ein Hersteller hat im Laufe der Zeit eine grosse Zahl von ähnlichen Produkten entwickelt. Diese weisen gleiche Topologien auf, unterscheiden sich also hauptsächlich in den Bemassungen. Irgendwann wird das Sortiment kategorisiert und für jede der dabei entstehenden Klassen eine parametrische Beschreibung entwickelt.
- Heute zunehmend wichtig ist die Entwicklung von Produkten mit CAD Systemen, welche eine Parametrisierung direkt unterstützen (zum Beispiel Pro/Engineer [LS93]). In einer solchen Umgebung entwickelt der Konstrukteur ein Produkt in welchem den konkret eingegebenen Bemassungen Parameter zugeordnet sind. Durch eine andere Belegung der Parameter kann später eine neue Ausprägung erzeugt werden. Damit lässt sich eine auf Bemassungen beschränkte Parametrik erreichen.
- Daneben findet sich aber auch eine sukzessive Entwicklung von Modellen. Die Neukonstruktion eines Produkts erfolgt ausschliesslich mit konkreten Werten. Wenn später ein ähnliches Produkt benötigt wird, so ersetzt der Konstrukteur konkrete Zahlenwerte durch algebraische Ausdrücke und algorithmischen Kode, bis das Design zum Schluss weitgehend parametrisiert ist.

Als Beispiel einer parametrischen Instanz kann die Beschreibung von Schraubenfedern dienen [LS93]. Deren Geometrie lässt sich durch die drei Kennlinien, Helixradius, Ganghöhe und Durchmesser des Federdrahtes bestimmen. Ein vereinfachtes mathematisches Modell für die Oberfläche liefert die nachfolgende Funktion $F(\psi, \varphi)$.

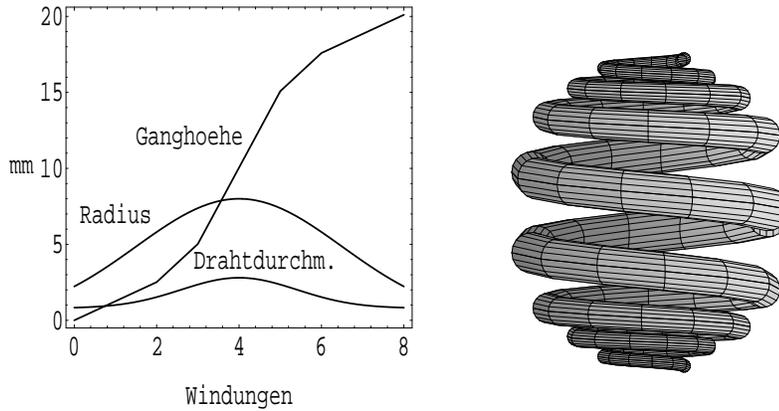


Abbildung 2.6: Ausprägung einer Schraubenfeder nach math. Modell

$$T(\varphi) = \begin{pmatrix} \cos(\varphi) & \sin(\varphi) & 0 \\ -\sin(\varphi) & \cos(\varphi) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$F(\psi, \varphi) = T(\varphi) * \begin{pmatrix} 0 \\ a(\varphi) + r(\varphi) * \cos(\psi) \\ r(\varphi) * \sin(\psi) \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ z(\varphi) \end{pmatrix}$$

$r(\varphi)$: Drahtradius, $a(\varphi)$: Helixradius, $z(\varphi)$: Ganghöhe

Durch konkrete Wahl der Parameter Drahtradius, Ganghöhe und Helixradius findet sich eine Ausprägung (siehe Abb. 2.6). In der Praxis sind aber die definierenden Kennwerte meistens mit Tabellen spezifiziert, ohne dass ein offensichtlicher funktionaler Zusammenhang zwischen der Anzahl Windungen und der Kennlinien besteht (siehe Abb. 2.7).

Ein anderes Beispiel für die Verwendung von parametrisch definierten Objekten im Datenaustausch findet sich bei Teilebibliotheken. Um die Zahl der in einem solchen elektronischen Katalog gespeicherten Daten zu reduzieren, werden diese parametrisch abgelegt. Der Konstrukteur konkretisiert Objekte später bei deren Verwendung.

Als Beispiel dafür ist in Abbildung 2.8 eine Darstellung einer Schraube mit möglichen Parametern gegeben. Nur ein parametrischer Ansatz erlaubt die Sortimentsvielfalt der Normteile elektronisch beherrschen zu können.

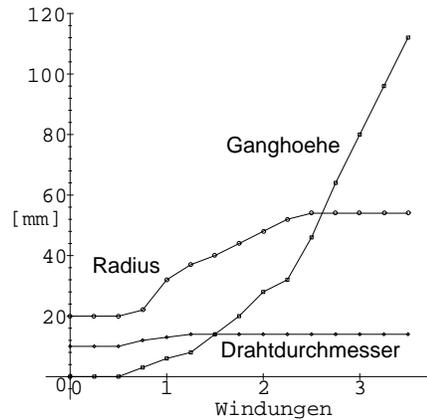


Abbildung 2.7: Drei Kennlinien einer Schraubenfeder

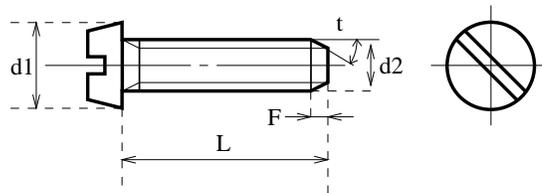


Abbildung 2.8: parametrisch definierte Schraube

existierende Lösungsansätze

Die schon sehr früh in IGES [SW86, IGE87, Smi91] definierten Makromöglichkeiten konnten sich erstaunlicherweise nicht durchsetzen. Da keine Implementation diese Eigenschaft unterstützte, wurde sie als nicht implementationsnotwendig zurückgestuft [IGE91]. Somit gibt es gegenwärtig kein standardisiertes Hilfsmittel zur Modellierung von Produktdaten, das die Parametrik unterstützt.

Etwas verschieden ist die Lage auf dem Gebiet der Programmiersysteme. So unterstützen verschiedene Systeme [Coc90, Mat93, MABD88] parametrische Eigenschaften, doch fehlt diesen Systemen die Möglichkeit, ein Datenmodell zu definieren und die persistent gespeicherten Daten auszutauschen.

Betrachtet man das Gebiet der Datenbanken, so stellt sich das Problem genau umgekehrt. Diese Systeme unterstützen meist die Spezifikation des Datenmodells, können aber keine parametrisch definierten Objekte bearbeiten. Langfristig muss es also zu einer Verbindung der bis heute getrennt gehaltenen Funktionalitäten kommen.

Parallel zu STEP gibt es unter der Bezeichnung “Part Libraries” (PLIB, ISO 15384) einen assoziierten Standard. In diesem wird versucht, ein Datenmodell für Teilbibliotheken zu definieren, das im Rahmen von STEP verwendbar ist. Wie schon dargestellt, sind parametrische Datenmodelle in diesem Bereich wichtig, und so musste die Gruppe nach einer eigenen Lösung suchen. Diese fand man in der Definition eines Datenmodells für algebraische Ausdrücke [Wie94, AA94]. Das Datenmodell weist allerdings die folgenden Nachteile auf:

- algebraische Ausdrücke werden nur sehr beschränkt unterstützt, algorithmische Sequenzen überhaupt nicht.
- bereits einfache Ausdrücke führen im Physical File zu einem komplexen Instanzengraph, da jeder Baustein einzeln instanziiert werden muss.
- die Loslösung des Modells von den “Integrated Resources” ermöglicht die Verwendung verschiedener, ähnlicher Lösungen. Damit ist eine standardübergreifende, einheitliche Verwendung des Modells nicht mehr garantiert. Dies verunmöglicht die Interoperabilität zwischen den Datenmodellen und kann im nachhinein kaum noch harmonisiert werden [SK92].

Trotz all dieser Probleme muss man sich bewusst sein, dass nur PLIB einen Ansatz für die Integration von Parametrik im Datenaustausch bietet. Weder die von der Object Management Group (OMG) entwickelte Object Management Architecture (OMA), noch die unter der Object Database Management Group (ODMG) entworfene, neue Objektdefinitionssprache (Object Definition Language, ODL [C⁺94, Cat95]) unterstützen die Modellierung von parametrischen Daten. Mit diesen neueren Vorschlägen, wie auch mit der kürzlich populär gewordenen Sprache Java [Har96], lassen sich verteilte Objekte definieren. Solche Definitionen bestehen aus Strukturbeschreibungen *und* der Spezifikation von Funktionssignaturen, respektive der Methoden selbst in Java. Die Idee eines eigentlichen Gesamtdatenschemas fehlt den erwähnten Ansätzen hingegen, und so können verteilte Objekte zwar aktiv verwendet werden, eine langfristige Archivierung in einem maschinenneutralen Format ist jedoch nicht möglich.

2.3.2 Sekundäre Klassifikationen

Objektorientierte Datenmodelle erzeugen eine hierarchische Typstruktur zwischen den abgeleiteten Datentypen. Damit kann diese Unterklassenbildung auch als Ausdruck einer Primärklassifizierung betrachtet werden. Polymorphismus erweist sich als grosser Vorteil für die Wiederverwendbarkeit von Definitionen, indem Unterklassen automatisch kompatibel zu ihren Vorgängern sind.

Allerdings verlangen heutige Anwendungen die Definition von sekundären Kriterien nach denen evaluierte Entitäten klassifiziert werden können. (Einige Forscher bezeichnen diese Eigenschaft als “Rollen”, die sich parallel zu den Datentypen definieren lassen.)

Beispiel: in geometrischen Datenmodellen unterscheiden sich Vektoren nicht nur bezüglich ihrer Dimension, sondern auch durch ihre Transformierbarkeit. Diese Kriterien sind orthogonal zueinander.

Sekundäre und allenfalls tertiäre Kriterien sind oft orthogonal zueinander. Sie lassen sich nur schlecht in die Primärkriterienstruktur integrieren, wenn gleichzeitig eine Typproliferation vermieden werden soll.

Zudem gilt zu beachten, dass die Veränderung der Klassenzugehörigkeit von evaluierten Instanzen eine sehr dynamische Angelegenheit ist. Auch soll die Objektidentität (OID), die zur eindeutigen Bezeichnung einer evaluierten Instanz dient, bei einem Rollenwechsel *nicht* verändert werden.

Eine ähnliche Situation entsteht, bei der Versionenbildung. Objektorientierte Modelle unterstützen durch die einfache Definition von Unterklassen die häufige Anpassung und Veränderung der Datenmodelle [BKkk87]. Versionen können als sekundäres Klassifikationsmerkmal betrachtet werden. Dementsprechend muss deren Dokumentation und Interoperabilität mit bestehenden Instanzen gewährleistet werden.

existierende Lösungsansätze

Die ersten Versuche mit zusätzlichen Klassifikationskriterien wurden in objektorientierten Programmiersprachen wie Self [Dav87, ABC⁺94], Smalltalk [GR83] oder Lisp [Kee89] gemacht. Allerdings verlangen all diese Sprachen aufwendige dynamische Typentests. Das ist sicher mit ein Grund, weshalb sich die Sprachen nicht durchsetzen konnten.

Auf theoretischer Seite ist ein Vorschlag in [Cha93] zu erwähnen. In dieser Konferenzpräsentation werden Datentypen vorgeschlagen, die Attribute kennen, welche nur dann ansprechbar sind, wenn assoziierte Prädikate erfüllt sind.

Ebenfalls erwähnenswert ist ein Vorschlag von Gottlob [GSR96]. Ausgehend von der Erkenntnis, dass ein starres Typsystem die Datenmodellierung zu stark einschränkt, werden Rollen als Lösung vorgeschlagen. So hat jede Instanz einen Typ und kann zusätzlich unterschiedlichen Rollen genügen. Der Ansatz, dass ein Objekt nach seinen rollenspezifischen Eigenschaften partitioniert wird, und dass jeder Teil eine eigene Objektidentität erhält, ist in der Produktdatenindustrie allerdings kaum realisierbar. Eine brauchbare Lösung muss berücksichtigen, dass sekundäre Klassifizierungsmerkmale nur Sichten auf eine Instanz darstellen; an diese kann *keine* eigene Objektidentität vergeben werden, ohne dass die Bedeutung der Identität und das Typsystem vollständig neu definiert werden.

Zur Veranschaulichung wie das Problem bis anhin behandelt worden ist, wird das Beispiel in Abbildung 2.9 (dazugehöriger SQL-Text in Abb. 2.10) verwendet. Die primäre Klassifizierung resultiert in den beiden Datentypen "TechPlan" und "StatusInfo". Jede technische Zeichnung kann zudem nach einem sekundären Kriterium, einem Zustand in der Entwicklungskette klassifiziert werden. So sollte jede Zeichnung zuerst gezeichnet, dann geprüft und schlussendlich für die

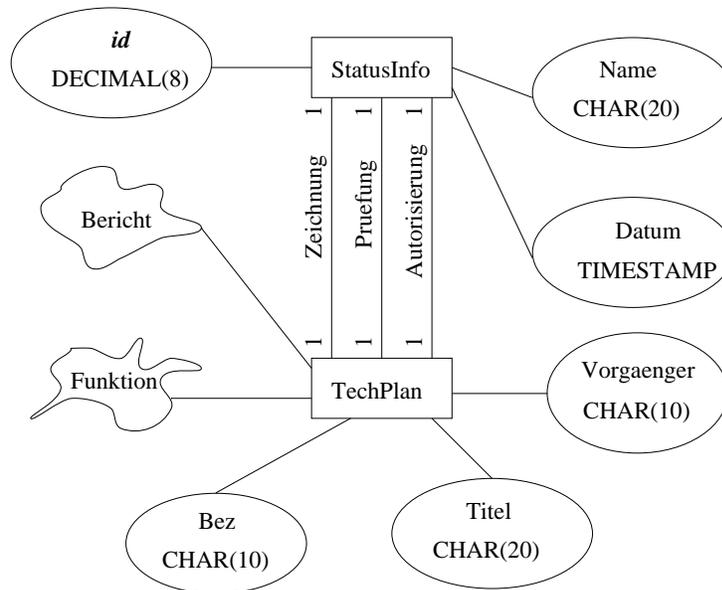


Abbildung 2.9: SQL-Modell, Dokumentverwaltung

Produktion autorisiert werden.

Sekundäre Klassifizierungen lassen sich in *relationalen Datenbanken* nur mit Hilfe von Attributen abspeichern. Damit ergeben sich mehrere Probleme, von denen die wichtigsten nachfolgend aufgeführt sind.

- Attribute mit sekundären Kriterien unterscheiden sich nicht von solchen die reine Informationsträger sind. Daher ist die Semantik von Klassifizierungsattributen nicht bekannt. (Als Folge kann die Sichtbarkeit von Attributen eines Tupfels kaum in Abhängigkeit eines Zustandes eingeschränkt oder erweitert werden.)
- Die Verwendung der Basisdatentypen (numerische Werte, Zeichen, Bitfolgen [DD93]) verhindert kontrollierte Wertebereiche für die möglichen Zustände.
- Kombinationen von Sekundärklassifizierungen lassen sich nicht einschränken, desgleichen sind Zustandsübergänge nicht kontrollierbar.

Als Konsequenz erkennt man, dass die gewünschten Eigenschaften vom System her nicht unterstützt werden. Eine Lösung lässt sich mit Konventionen erreichen, doch sind solche Abmachungen und ihre Bedeutung kaum je im Datenmodell dokumentiert. Daher ist eine langfristig sichere Methode nicht gegeben.

Eine Eigenschaft von objektorientierten Datenbanken ist, dass jeder Instanz nebst einem Typ auch ein Objektidentifikator (OID) zugeordnet ist. Dieser dient der eindeutigen Bezeichnung eines jeden aktiven Objekts. Damit kann sich

```

CREATE TABLE TechPlan
(
  Bez          CHAR(10),
  Titel        CHAR(30),
  Vorgaenger   CHAR(10),

  Zeichnung    DECIMAL(8),

  Pruefung     DECIMAL(8),
  Bericht      bericht,

  Autorisierung DECIMAL(8),
  Funktion     funktion,

  PRIMARY KEY (Bez),
  FOREIGN KEY (Vorgaenger)
    REFERENCES TechPlan,
  FOREIGN KEY (Zeichnung)
    REFERENCES StatusInfo,
  FOREIGN KEY (Pruefung)
    REFERENCES StatusInfo,
  FOREIGN KEY (Autorisierung)
    REFERENCES StatusInfo)
)

CREATE TABLE StatusInfo
(
  id          DECIMAL(8),
  Name       CHAR(20),
  Datum      TIMESTAMP,

  PRIMARY KEY (id)
)

```

Abbildung 2.10: SQL Modell, techn. Zeichnung

der Typ eines Objektes grundsätzlich ändern, doch muss mit einem aufwendigen Prozess die referentielle Integrität bezüglich des OID sichergestellt werden. Ein Vorschlag für die Lösung des Dokumentverwaltungsproblems findet sich in [CLR93], indem verlangt wird, dass zu jedem Zeitpunkt alle Entwurfsalternativen vorhanden sein müssen. Diesen Ansatz hat man auch in STEP [ISO94d, ISO94e, ISO94f] gewählt. Als Konsequenz entsteht eine anonyme Entität, an die nun baumartig einzelne Instanzen gebunden werden. Dieser Ansatz erlaubt jedoch keine typsichere Behandlung der Unterbäume und erschwert die Zustandskontrolle im Modell. Andererseits entfallen aber die aufwendigen Datenbanktests zur Sicherstellung der referentiellen Integrität.

Ein anderer Ansatz wird mit der Verwendung von Unterklassen gemacht. Sowohl in *EXPRESS* als auch in Galileo [Ghe90] und in UNISQL/M [KGK⁺95] sind Instanzen durch sämtliche zu einer Zeit gültigen Datentypen bestimmt. Das führt in *EXPRESS* zu einem Datenmodell wie in Abbildung 2.11 dargestellt. Der Nachteil dieses Ansatzes beruht darin, dass *EXPRESS* keine Veränderung der Typzugehörigkeit seiner Instanzen zulässt, und dass die Definition der gültigen Klassifizierungskombinationen statisch ist und zu sehr komplizierten Bedingungen führt (siehe textuelle Definition von “TechPlan” in Abb. 2.12).

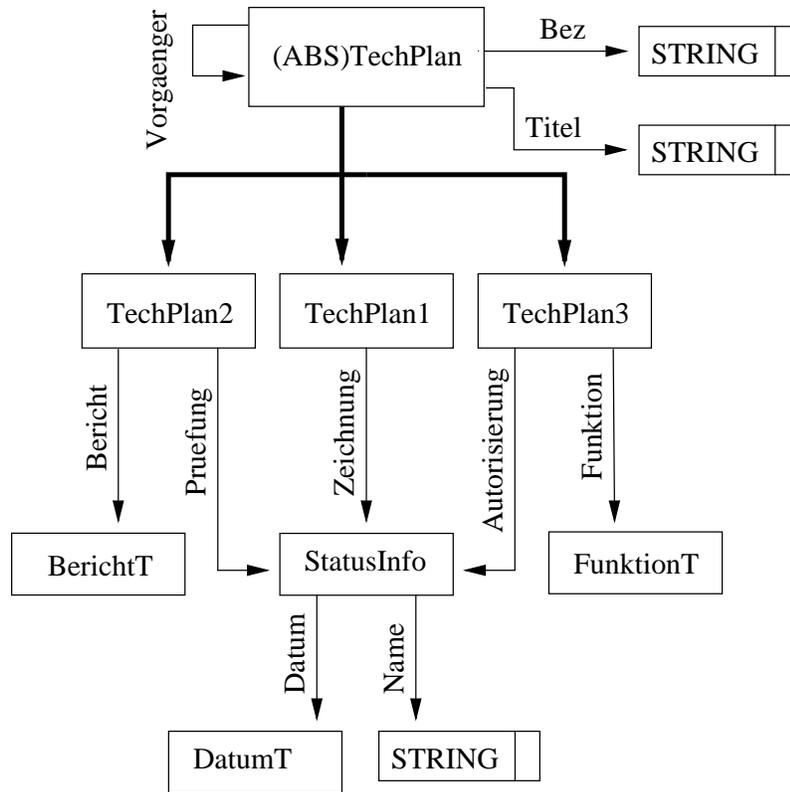


Abbildung 2.11: EXPRESS-Modell, Dokumentverwaltung

So wird in einem nächsten Schritt ein dynamisches Setzen von sekundären Kriterien ermöglicht, wie das in EXPRESS-C [SNS94] und Galileo vorgeschlagen wird. Ebenfalls ist ein noch weitergehender Schritt denkbar, indem sich wie zum Beispiel in COCOON [SLR⁺92] die erlaubten Typkombinationen (und damit kombinierbaren Klassifizierungen) nicht mehr innerhalb der primären Typhierarchie bewegen müssen. Dynamisch lassen sich weitere Kriterien an evaluierte Entitäten binden. Der grösste Nachteil dieses Ansatzes liegt in den fehlenden Dokumentationsmöglichkeiten der erlaubten Kombinationen im Datenmodell. Daneben muss aber auch bei jeder Dereferenzierung ein Typentest erfolgen, was zu einer bemerkbaren Leistungseinbusse führt. Beachtet man auch, dass die Mehrheit der Instanzen in ihren Klassifizierung stabil bleiben, so ist ersichtlich, dass in vielen Fällen ein effizienteres Verfahren denkbar sein muss.

2.3.3 Operationelle Eigenschaften von Daten

In der Typtheorie wird heute allgemein anerkannt, dass eine vollständige Typdefinition aus einem strukturellen und einem operationellen Teil besteht [Löf84, BCM88]. Dabei beinhaltet der operationelle Teil eine Sammlung von Basisfunktionen zur Festlegung der Grundfunktionalität des Datentyps.

```

ENTITY StatusInfo;
  Name:    STRING(10);
  Datum:   DatumT;
END_ENTITY;

ENTITY TechPlan ABSTRACT SUPERTYPE OF
  (TechPlan1 ANDOR (TechPlan1 AND
  (TechPlan2 ANDOR (TechPlan2 AND TechPlan3))))
  Bez:     Label;
  Titel:   STRING(30);
  Vorgaenger: TechPlan;
END_ENTITY;

ENTITY TechPlan1 SUBTYPE OF (TechPlan);
  Zeichnung: StatusInfo;
END_ENTITY;

ENTITY TechPlan2 SUBTYPE OF (TechPlan);
  Pruefung: StatusInfo;
  Bericht:   BerichtT;
END_ENTITY;

ENTITY TechPlan3 SUBTYPE OF (TechPlan);
  Autoris:   StatusInfo;
  Funktion:  FunktionT;
END_ENTITY;

```

Abbildung 2.12: Modell, Zeichnungsverwaltung

In der Datenmodellierung wurde der Schritt zu einer mit operationellen Definitionen verfeinerten Spezifikation von Modellen noch nicht vollzogen. Zum einen fehlt es an einem einheitlichen Modelliermittel für algorithmische Abläufe, zum anderen werden Algorithmen in der Datenbankwelt immer noch stark von den Daten und Datendefinitionen getrennt.

In *EXPRESS* ist man vermutlich am weitesten fortgeschritten, indem die Modellersprache zumindest das Mittel des abgeleiteten Attributes (“derived attribute”) kennt. Dabei handelt es sich um ein methodenähnliches Konstrukt, das gleich einem Attribut mit Objekten assoziiert ist und auch vererbt wird. Eine explizite Parameterübergabe ist nicht möglich, was die Verwendbarkeit der abgeleiteten Attribute stark einschränkt. Der Grund dafür liegt in der Ansicht der *EXPRESS*-Entwickler, dass sämtliche prozeduralen Teile eine *statisch* definierbare Eigenschaft eines Datenmodells darstellen. Daher werden keine algorithmischen Kodeteile ausgetauscht, und Methoden als dynamische Elemente erhalten in *EXPRESS* keine Unterstützung.

Gesucht wird eine speziell für die Standardisierung interessante Eigenschaft, die zur Laufzeit das Anbinden von prozeduralem Code an Instanzen erlaubt. Oftmals weiss der Modellierer nämlich, dass eine operationelle Eigenschaft exi-

stiert, ohne dass er sie bereits zu modellieren vermag. Für diesen Fall ist es notwendig, deren Existenz im Datenmodell zu dokumentieren, die eigentliche Implementierung aber bis zur Laufzeit aufschieben zu können.

existierende Lösungen

Wie bereits erwähnt, unterstützen die heutigen Datenmodelliermethoden keine algorithmischen Teile. Zwar kennt SQL2 [DD93] sogenannte "Trigger", die bei gewissen Ereignissen aktiviert werden, doch hat sich dieser Standard in der Praxis nicht durchgesetzt. Wieweit sich diese oder die für die nächste Version des Standards geplanten prozeduralen Teile im Datenbankbereich etablieren können ist daher fraglich.

Einige Anbieter von relationalen Datenbanken, wie zum Beispiel Sybase, kennen sogenannte "stored procedures". Diese bestehen aus einem in der Datenbank abgelegten Bytecode, der von der Anwendung aus aufgerufen werden kann. Dabei ist jedoch keine Vererbung der Prozeduren oder explizite Parameterübergabe aus der Applikation möglich.

Die hilfreichsten Ansätze finden sich erneut im Bereich der objektorientierten Programmiersprachen und besonders in den Systemen, die eine persistente Datenverwaltung unterstützen. Allerdings konzentrieren sich die meisten objektorientierten Programmiersprachen auf die Kompilation von Quelltext in einen direkt für die ausführende Maschine zugeschnittenen Objektcode. Zusätzlich ist das dynamische Nachladen von einzelnen Modulen sehr stark eingeschränkt und kann daher auch nicht zur Erweiterung der Funktionalität von Daten verwendet werden. Eine erwähnenswerte Ausnahme bildet dabei das Oberon System [WG92], das zwar hauptsächlich mit in Maschinensprache übersetztem Kode arbeitet, diesen aber nach [Fra94] dynamisch erweitern kann. Weitere Ausnahmen stellen all diejenigen Systeme dar, die eine stark interpretative Komponente aufweisen, also zum Beispiel Euklid im CAD-Bereich und Prolog, Scheme [AS93], Lisp [Kee89, Pae93] und Self [Dav87] bei den reinen Programmsystemen. Ein genereller Nachteil bei Programmiersystemen besteht aber meist in den eingeschränkten Dokumentationsmöglichkeiten, respektive den system-spezifischen Eigenheiten, die eine allgemeine Verwendung der Definitionen in anderen Systemen nicht zulassen.

2.4 Vergleich von Datenmodelliermitteln

Der Vergleich von verschiedenen Datenmodellierkonzepten gestaltet sich schwierig, da

- Begriffe mit sehr verschiedenen Bedeutungen verwendet werden. So kommt es vor, dass gleiche Konzepte unter verschiedenen Namen bekannt sind, oder dass dieselben Begriffe völlig unterschiedliche Bedeutungen annehmen.

- Datenmodellparadigmen mit Blick auf ein Datenmodell zur Implementation entwickelt wurden. Bedeutendster Vertreter dieser Art ist das Entity-Relationship-Modell. Es geht faktisch von einer Umsetzung auf relationale Datenbanken aus.
- unterschiedliche Anforderungen für den Einsatz der Modellieretechnik existieren.

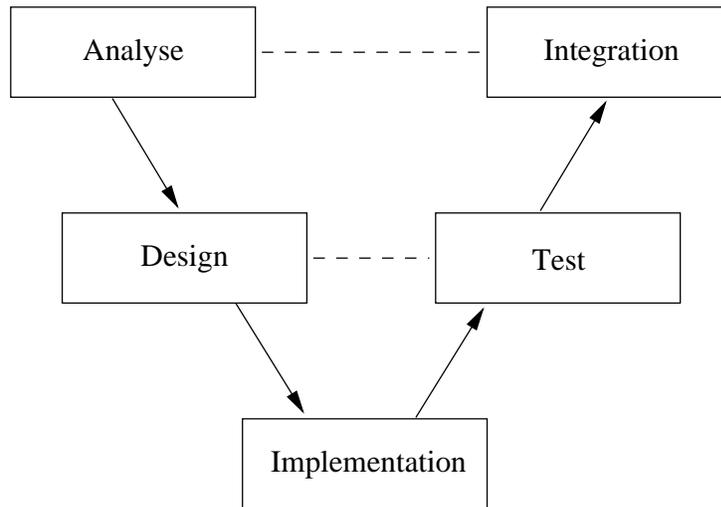


Abbildung 2.13: Entwicklungsstufen für eine Anwendung

Besonders der letzte Punkt ist erwähnenswert. Abbildung 2.13 stellt einen typischen Verlauf für die Entwicklung einer Anwendung dar. Dazu ist zu bemerken, dass die Datenmodelliersprache erst auf der zweiten Stufe (“Design”) eingesetzt wird. Sie befindet dadurch an der Schnittstelle zwischen Implementation und Problemanalyse, die oft mit einem eigenen Hilfsmittel vorgenommen wird. Das bedeutet aber, dass die Hilfsmittel zur Datenmodellierung sich meist den Anforderungen aus den umgebenden Schichten anpassen.

Speziell in der Standardisierung ergeben sich zusätzliche Anforderungen an die Modellieretechniken. Zum einen darf ein Datenmodell nicht derart spezifisch ausfallen, dass die Implementationsstufe sowohl in der Wahl der Programmiersprache, als auch in den Algorithmen festgesetzt wird. Zusätzlich sind auch Iterationen zur Datenmodellverbesserung und -entwicklung nur beschränkt möglich. Während dies in den Anwendungen (gegeben durch die horizontalen, unterbrochenen Linien in der Abbildung) dazu dient die Systeme iterative zu verbessern, wird ein solches Vorgehen in der Standardisierung durch die demokratische Entscheidungsfindung und Publikationsvorschriften erschwert.

Im folgenden werden einige oft in Projekten eingesetzte Modellieretechniken sowie ein neueres Datenmodell aus der Datenbankforschung mit *EXPRESS* verglichen. Um die genannten Probleme beim Vergleich zu eliminieren, wurde versucht Kriterien festzulegen, die für die Anwendung in Produktdatenmodellen von hoher Komplexität sinnvoll sind. Die verglichenen Eigenschaften sind somit:

- graphische/textuelle Notation,
- Konsistenzbedingungen an Typen und Daten,
- Funktionen oder Methoden als Teil des Datenschemas,
- dynamische Eigenschaften und die
- Modularisierungshilfen

Damit ist ein verhältnismässig neutraler Vergleich möglich, ohne dass auf die fanatisch geführten Streitgespräche der Anhänger einzelner Methoden eingegangen werden muss.

2.4.1 Entity Relationship

Das Entity-Relationship (ER) Modell wurde 1976 von Chen [Che76] publiziert. Es wird hier einzig wegen seiner Popularität erwähnt. Basiselemente des Modellierkonzepts sind Entitäten mit Attributen sowie Relationen zwischen Entitäten. Letztere sind als “existierende Objekte, die sich von anderen Objekten unterscheiden lassen” [KS86] definiert. Das System basiert auf einer etablierten Theorie. Zumeist wird eine Implementation unter Verwendung von relationalen Datenbanksystemen angestrebt. Das zeigt sich unter anderem in der sehr frühen Wahl von Schlüsselkandidaten zusammen mit der expliziten Nennung von Kardinalitäten. Allerdings werden nur einfache Kardinalitätsbedingungen unterstützt. Weitere Konsistenzbedingungen ebenso wie Funktionen, dynamisches Verhalten oder bedingt sichtbare Elemente sind im ER-Modell nicht bekannt. Zu einem späteren Zeitpunkt wurde das Modell um die Möglichkeit der *Generalisierung* und *Spezialisierung* erweitert [SS77]. Damit sind rudimentäre Definitionsmittel zur Auszeichnung von Rollen vorhanden.

Ein genereller Vorteil des ER-Modells im Zusammenhang mit der Standardisierung liegt darin, dass sich Tabellen auf vielen Systemen direkt aus dem Modell erzeugen lassen. Zusätzlich wird der Zugriff und die Bearbeitung relationaler Daten für verschiedenste Programmiersprachen unterstützt. Damit sind die Modelle von ihrer Implementationsunabhängigkeit her geeignet. Andererseits bewirkt das Fehlen von Vernetzungselementen und Aggregationen, dass Produktdatenmodelle nur umständlich und mit Verzicht auf formale Konsistenzbedingungen spezifizierbar sind.

2.4.2 Rumbaugh

Die “Object Modeling Technique” (OMT) [RBP⁺91] wurde vor etwa sechs Jahren konzipiert und zielt, wie der Name andeutet, auf objektorientierte Datenmodelle ab. Die Technik nimmt für sich in Anspruch, sowohl den Analyse- als auch den Entwicklungsprozess unterstützen zu können. Eine Umsetzung in ein beliebiges objektorientiertes System sollte möglich sein.

Die Grundideen von OMT sind denjenigen von *EXPRESS* sehr ähnlich, und so gibt es Klassen mit Attributen als Grundelemente. Zusätzlich lassen sich aber auch funktionale Aspekte in OMT-Klassen definieren. Ebenfalls vorhanden sind abgeleitete Attribute sowie die Möglichkeit, abstrakte Klassen bilden zu können. Die Generalisierung verwendet Vererbungskonzepte bei denen disjunkte und nichtdisjunkte Unterklassen definierbar sind. Diese führen zu einem ähnlichen Ergebnis, wie in *EXPRESS*. Allerdings ist dessen Supertypeexpression mächtiger.

Im Gegensatz zu *EXPRESS* ist in OMT die explizite Spezifizierung von Relationen vorgesehen. Die Entwickler von *EXPRESS* haben nach langen Diskussionen nicht erkennen können, wann ein Informationselement als Attribut zu modellieren, und wann es besser als Beziehung darzustellen ist. Aus diesem Grund ist man im Rahmen von STEP davon abgekommen, das Beziehungskonzept zu unterstützen. Andererseits gibt es immer wieder Gruppierungen, die diese Eigenschaft in Modellersprachen integriert sehen wollen.

Zusammenfassend kann man sagen, dass OMT mit *EXPRESS* sehr viele Gemeinsamkeiten aufweist. Die vorwiegend graphisch orientierte Modellieretechnik basiert auf objektorientierten Prinzipien, kennt ein Modulkonzept und erlaubt zusätzlich die Spezifikation von dynamischen Abläufen mit Hilfe von Zustandsübergangsdiagrammen. Etwas schwächer ist die Technik in den Bereichen der Konsistenzsicherung (kennt nur Kardinalität und abstrakte Klassen), der Typenkombination (kennt nur disjunkte, nichtdisjunkte Partitionierung) und der textuellen Definitionsmöglichkeiten.

2.4.3 Coad/Yourdon

Coad und Yourdon haben sich bereits mit der Herausgabe einer Modellieretechnik zur Analyse und Design für relationale Datenysteme einen Namen gemacht. Ihre Vorschläge kommen aus der Praxis, und so ist es nicht verwunderlich, dass später ein weiterer Ansatz unter Berücksichtigung von objektorientierten Grundlagen entwickelt wurde [CY94, CY91].

Da die Entwicklungsphilosophie der Autoren nicht deckungsgleich ist, mit derjenigen aus Abbildung 2.13 (s. Seite 23), müssen beide Techniken, sowohl diejenige für die Analyse als auch die für die Designspezifikation zusammen betrachtet werden. Dabei wird klar, dass sich das Modellierungsprinzip an verschiedenen Stellen noch an relationalen Datenbanken orientiert. So etwa wenn im Designteil "Schlüssel und abgeleitete Attribute eingeführt, sowie Normalisierungen angestrebt" werden sollen.

Andererseits beruht die Modellieretechnik aber auch auf objektorientierten Prinzipien und kennt den Klassenbegriff synonym zur Entität in *EXPRESS*. Ebenfalls bekannt ist die Spezialisierung mit den beiden Fällen der Ein- und Mehrfachvererbung. Im Gegensatz etwa zu *EXPRESS* sind Kombinationstypen explizit durch Mehrfachvererbung zu modellieren. Dies würde besonders in Produktdatenmodellen zu einer enormen Komplexität der Modelle führen. Auch in der Standardisierung dürfte das problematisch werden, da eine explizite Model-

lierung aller Kombinationstypen verlangt, dass alle Anwendungsprotokolle zur selben Zeit standardisiert werden. Andernfalls muss das Datenschema in einem iterativen Prozess angepasst werden, was in der Standardisierung aufwendig und zeitraubend ist.

Nebst Klassen mit Attributen werden aber auch die Modularisierung, die Aggregation, methodenartige Konstrukte (unter dem Namen “services”) und Assoziationen unterstützt. Jedoch dienen die Kardinalität und abstrakte Klassen als einziges Mittel zur Konsistenzspezifikation. Speziell an dem Modellierprinzip ist die Idee, die Abhängigkeit von Methoden zwischen den beteiligten Klassen auszuweisen.

Alles in allem wirkt diese ausschliesslich graphikbasierte Modellertechnik älter und wenig geeignet zur Verwendung in Produktdatenmodellen in der Standardisierung. Dies ergibt sich aus

- dem Fehlen von komplexeren Konsistenzbedingungen
- den ungenügenden Referenzierungshilfen zwischen Modulen
- und der erwähnten Unfähigkeit, überlappende Klassen auf einfache Art bilden zu können.

2.4.4 Booch

Die Entwicklungsmethode von G. Booch [Gra91] ist ebenfalls ein industriell eingesetztes Hilfsmittel auf objektorientierter Basis. Das Resultat eines Analyse- und Designprozesses zeigt sich in mehreren graphischen Modellen wobei folgende Diagramme zu Verfügung stehen:

- **Klassendiagramm** zur Darstellung der strukturellen Eigenschaften von Klassen.
- **Zustandsübergangendiagramm** zur Definition von Zuständen und Ereignissen in einem System.
- **Objektdiagramm**, das die Eigenschaften von Objekten mit ihren wechselseitigen funktionalen und strukturellen Eigenschaften aufzeigt.
- **Interaktionsdiagramm** zur graphischen Modellierung von Interaktionen zwischen Objekten.
- **Moduldiagramm**, das die Module und ihre Schnittstellen aufzeigt sowie ein
- **Prozessdiagramm**, das die Zuordnung zwischen Systemen und verschiedenen Prozessen darstellt.

Die Zahl dieser Diagramme ist eine Folge der Partitionierung der Entwicklungsanforderungen gemäss Bild 2.14. Bedenkt man, dass zu jedem Modell eine

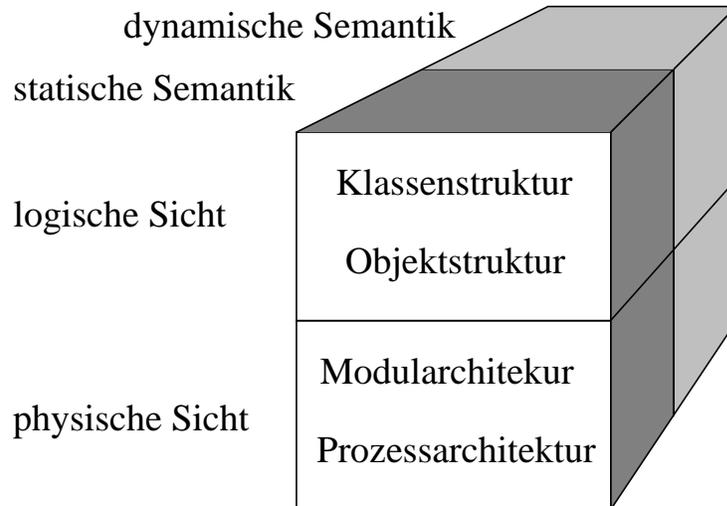


Abbildung 2.14: Modelle der objektorientierten Entwicklung [Gra91]

Zahl von Notationen hinzukommt, so wird die Komplexität des Modelliermittels klar.

Von seinen Eigenschaften unterscheidet sich diese Methode kaum von anderen Vorschlägen. So sind Attribute, Assoziationen, Aggregationen, Methoden und Subtypenbildung mit Vererbung bekannt. Zusätzlich existieren die Modelliermittel der Metaklassen und der Templates, die aber beide Konzepte von wenigen Programmiersystemen darstellen.

Bei der Vererbung ist nur die von den Programmiersprachen bekannte normale und mehrfache Vererbung unterstützt. Daher müssen auch in diesem System Klassenkombinationen explizit durch Mehrfachvererbung modelliert werden.

Im Bereich der Konsistenzdefinitionen kennt die Methode von Booch nur gerade Kardinalitäten, die aber auch mit Hilfe von relationalen Operatoren ausgedrückt werden können.

Zusammenfassend zeigt sich, dass diese Methode mehr Möglichkeiten aufweist als in der Standardisierung von Produktdatenmodellen (momentan) benötigt werden. Andererseits sind in den anwendbaren Teilen der logischen Sicht (siehe Abb. 2.14), zuwenige Eigenschaften vorhanden, um die Datenmodelle mit ausreichender Semantik zu versehen.

Im Bereich der kommerziellen Datenmodelliermittel zeichnet sich die Entstehung einer Kombination aus den Vorschlägen von Rumbaugh, Booch und Jacobson ab. Die daraus entstehende "Unified Modeling Language" wurde mit vielen Ideen aus anderen Methoden angereichert und soll den gesamten Softwareprozess von der Analyse bis zu Implementation unterstützen.

2.4.5 Das OM Datenmodell von Norrie

Das von Norrie entwickelte Objektmodell [NSWW96] zeigt die wesentlichen Eigenschaften, die für die Produktdatenmodellierung notwendig sind. Eine Implementation existiert [Wür95], doch ist das System nicht kommerziell verfügbar.

Die Datenmodellierung kann sowohl mit einer graphischen, als auch mit einer textuellen Notation erfolgen. Dabei ist das Datenmodell als Synthese aus dem Entity-Relationship Modell und objektorientierten Ansätzen zu verstehen. Damit werden sowohl explizite Beziehungen als auch Attributreferenzen zwischen Entitäten unterstützt, wobei jedoch letztere in der graphischen Darstellung nicht ausgewiesen werden.

Im Unterschied zu den meisten anderen Datenmodellen wurde als Grundaxiom die orthogonale Unterstützung von Typ und Klassifikation verfolgt. Daraus ergibt sich die Möglichkeit, Spezialisierungen kombinieren zu können. Dies erfolgt unter Verwendung von Konsistenzbedingungen zur genaueren Spezifikation der erlaubten Kombinationstypen. Die vorhandenen Kombinationsoperationen sind:

- **cover**, das dem ANDOR in *EXPRESS* sehr nahesteht und überlappende Typen ermöglicht.
- **intersect**, das genau dem AND in *EXPRESS* entspricht; damit werden überlappende Typen erzwungen.
- mit **disjoint** wird die Ueberlappung von Typen verhindert. Dies muss in *EXPRESS* als Ausdruck dargestellt werden.
- **partition** als vollständige Auftrennung in nichtüberlappende Typen, was in *EXPRESS* durch ONEOF ausgedrückt wird.

Ebenfalls bemerkenswert ist, dass dieses Datenmodell als einziges objektorientiertes Hilfsmittel eine formale Definition der Semantik kennt. Damit lässt sich mit dem Modell viel sicherer arbeiten, und auch Erweiterungen dürften einfacher, und vor allem widerspruchsfrei integrierbar sein. Bei allen anderen Datenmodellen, einschliesslich *EXPRESS* fehlt diese Grundlage, was immer öfter als Mangel erkennbar ist.

Gegenüber *EXPRESS* machen sich vor allem zwei fehlende Eigenschaften bemerkbar. So sind algorithmische Konsistenzbedingungen an die Daten nicht Teil des Modells, und weiter sind keine Modularisierungshilfen erkennbar. Bei der Spezifikation von sehr grossen Datenmodellen, wie etwa in der Auto- oder Flugzeugindustrie, sind diese beiden Eigenschaften jedoch zwingend. Allerdings dürfte eine dahingehende Erweiterung relative einfach ausfallen.

Kapitel 3

Parametrik

Wie in Kapitel 2.3 dargelegt, werden immer häufiger CAD Systeme mit parametrischen Eigenschaften eingesetzt. Damit ist auch klar, dass ein Bedarf für den Austausch von Daten zwischen solchen Systemen besteht. Heute fehlt eine derartige Möglichkeit auf nichtproprietärer Basis und die im Unterkapitel 2.3.1 vorgestellten, existierenden Lösungsansätze sind zu aufwendig und bieten gleichzeitig nur eingeschränkte Möglichkeiten.

3.1 Lösungsvorschlag

Der wichtigste Schritt zur Einführung parametrischer Datensysteme, liegt in der Erkenntnis, dass die gängigen Datenmodelliersprachen Zahlenwerte und nicht evaluierte Ausdrücke zu stark trennen. Modelle die auf solchen Paradigmen beruhen sind zwar direkt und rasch implementierbar. Von der Theorie her ist die Trennung von evaluierten und nicht evaluierten Ausdrücken jedoch künstlich und nicht notwendig. Aus diesem Grund muss die erste Forderung lauten

Zahlenwerte, Instanzen und noch zu evaluierende Ausdrücke sind beim Datenaustausch gleich zu behandeln.

Dies bedeutet eigentlich nichts anderes, als dass aus der Sicht des Modellierers keinen Unterschied besteht, ob Instanzen mit Hilfe von Zahlenwerten oder Ausdrücken gebildet werden. Zahlenwerte sind wie in symbolischen Rechensystemen (zB. in [Eng69]) als atomare Einheiten zu sehen, aus denen Ausdrücke aufgebaut werden können. Die Aehnlichkeit zu symbolischen Rechensystemen geht noch weiter, wenn man das Beispiel der Definition einer Schraubenfeder betrachtet. Die dargestellten Abhängigkeiten lassen sich im allgemeinen nur umständlich als mathematische Funktionen ausdrücken. Viel einfacher, und für die Designer naheliegender erweisen sich dazu algorithmische Kodestücke. Somit folgt als zweite Voraussetzung für den Austausch von parametrisch definierten Instanzen

algebraische Ausdrücke sind eine spezielle Form von Funktionen. Diese können auch aus algorithmischem Code bestehen und müssen als selbständige, referenzierbare Einheiten austauschbar sein.

Die Konsequenz dieser Forderung ist, dass Funktionen zu sogenannten “first-class-objects” gemacht werden müssen. Darunter werden Objekte verstanden, die gleich wie einfache Datenwerte zuweisbar, referenzierbar, vergleichbar und austauschbar sind.

Die beiden Forderungen lassen im Rahmen von STEP nur erfüllen, wenn zumindest das Austauschformat [ISO94c] erweitert wird. Zusätzlich erscheint es sinnvoll, bereits zu diesem Zeitpunkt ein methodenartiges Konzept einzuführen. Dazu wird ein explizites Konstrukt zur Deklaration von funktionswertigen Typen definiert. Funktionswertige Objekte grenzen sich von den bereits existierenden Typen dadurch ab, dass sie im Gegensatz zu den Wertetypen nur Funktionen referenzieren können. Der Unterschied zu den bereits in *EXPRESS* existierenden “derived attributes” besteht darin, dass die Werte von funktionswertigen Attributen nicht bereits zur Modellierzeit bekannt sind. Das bedingt, dass Funktionen im Austauschformat integriert und mitausgetauscht werden. Der neue Datentyp führt vorerst zu folgender Erweiterung der *EXPRESS*-Grammatik:

```
base_type := simple_type | aggr_type | func_type | id.
func_type := "FUNCTION" ftype_list ":" param_type.
ftype_list := '(' [param_type {',' param_type}] ')'
```

Diese Erweiterung verlangt für den Datenaustausch respektive für die Archivierung ein etwas aufwendigeres Laufzeitsystem mit einem integrierten Interpreter wie das in Abbildung 3.1 schematisch dargestellt ist.

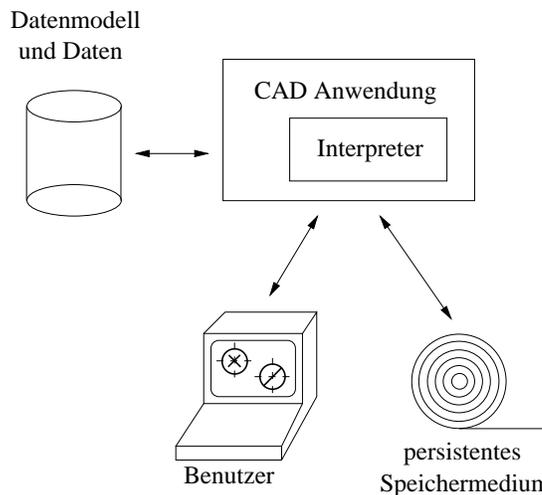


Abbildung 3.1: konzeptueller Austausch parametrischer Daten


```

-- nicht detailliert
HEADER := ... .
TRAILER := ... .

-- BINARY, AGGR, ENUM, INTEGER, REAL und STRING
-- definieren die Kodierung von Werten des je-
-- weiligen Datentyps.

stat := (astat | ifstat | forstat | whilestat |
         procstat | 'SKIP' | 'ESCAPE' |
         'RETURN '(' expression ')') ';'.
astat := designator {qualifier} ':=' expression.
ifstat := 'IF' expression 'THEN' {stat}
         ['ELSE' {stat}] 'END_IF'.
forstat := 'FOR' id ':=' expression
         'TO' expression ['BY' expression]
         {stat}
         'END_FOR'.
whilestat := 'WHILE' expression
         'DO' {stmt} 'END_WHILE'.
procstat := builtin_proc aparam_list.
aparam_list := '(' [expression {',' expression}] ')'.
fparam_list := '(' [fpar {',' fpar}] ')'.
ftype_list := '(' [param_type {',' param_type}] ')'.
fpar := id {',' id} ':' param_type.
param_type := base_type | general_type.
base_type := simple_type | aggr_type | func_type | id.
aggr_type := 'ARRAY' '[' expression ':' expression ']'
         'OF' ['OPTIONAL']['UNIQUE'] param_type |
         'SET' '[' expression ':' expression ']'
         'OF' param_type |
         'BAG' '[' expression ':' expression ']'
         'OF' param_type |
         'LIST' '[' expression ':' expression ']'
         'OF' param_type.
func_type := 'FUNCTION' ftype_list ':' param_type.
general_type := 'GENERIC' [':' id] |
         'AGGREGATE' [':' id] 'OF' param_type.
expression := factor {binop factor}.
factor := [unop] '(' expression ')' | primary.
primary := literal | designator.
designator := 'SELF' [qual1] |
*          REF [qual2] |
          id [qualifier].
qual1 := ('.' | '\') id [qualifier].
qual2 := ('.' | '\') id [qualifier]
         | aparam_list.
qualifier := { ('.' | '\') id |
              aparam_list |
              '[' expression ']' }.

```

Bei der Zusammenführung der beiden Grammatiken wurden einige Vereinfachungen vorgenommen. Diese betreffen vor allem die Erzeugung von arithmetischen Ausdrücken. Operatorpräzedenzen werden nicht wie in der Originalgrammatik durch eine Hierarchie der Syntaxregeln spezifiziert. Ebenfalls verzichtet wurde auf das *ALIAS*-Konstrukt, nachdem eine Analyse von 170 Datenmodellen zeigte, dass dieses Konstrukt in der Praxis keine Anwendung findet.

Aus theoretischen Gründen ist die Möglichkeit von lokalen Definitionen im Austauschfile stark eingeschränkt. Diese führen im allgemeinen zu Problemen beim Datenaustausch, da der Empfänger sie nur schwer verarbeiten kann. Eine gewisse Lokalität von Instanzen ist aber mit Hilfe des bereits existierenden *SCOPE*-Konstrukts [ISO94c] möglich.

Um den Integrationsvorgang der beiden Grammatiken besser verstehen zu können, sind alle Produktionen aus [ISO94b] mit Grossbuchstaben bezeichnet, während diejenigen aus [ISO94c] kleingeschrieben sind. Im weiteren sind die Einstiegspunkte für die folgenden Erläuterungen mit einem Stern (*) markiert.

Das Austauschfile besteht schon im existierenden Standard aus mehreren Teilen, einem Vorspann mit allgemeinen Informationen, den eigentlichen Austauschdaten gefolgt von einem Endbezeichner. Letzterer dient der eindeutigen Markierung des Endes des Datenfiles. Neu kommt nach dem Vorspann ein optionales Element zur Bezeichnung freier Variablen. Mit Hilfe der Produktion "PARAM_SECTION" werden freie Variablen mit ihrem Namen und Typ vorgestellt. Im nächsten Kapitel wird mit Hilfe der formalen Regeln gezeigt, dass die Nennung von freien Variablen eigentlich nicht notwendig ist. Diese Information dient jedoch dem empfangenden System als Hilfe um frühzeitig zu erkennen, ob parametrische Daten erwartet werden, und gleichzeitig um eine Konsistenzprüfung der Datentypen von Beginn weg zu ermöglichen.

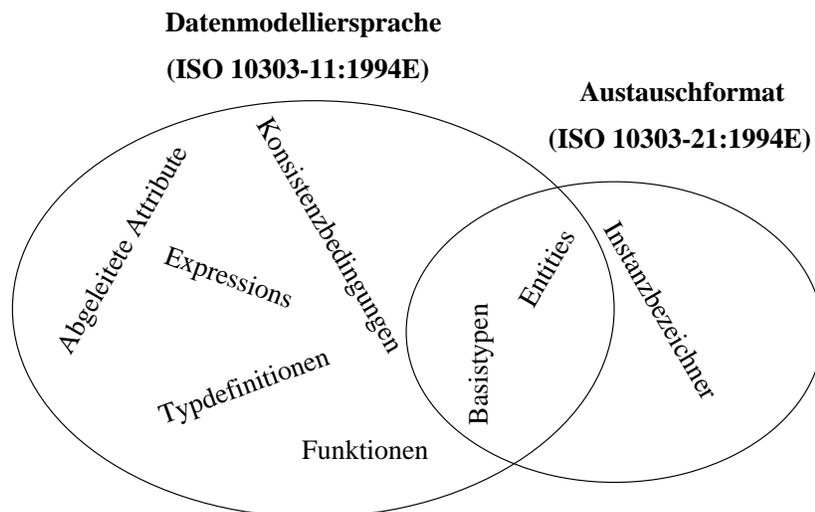


Abbildung 3.4: Datendefinitionssprache/Austauschmodell alt

Auch neu respektive modifiziert sind die Syntaxregeln “FUNCTION_INST” und “INSTANCE”, die eine Definition von referenzierbaren Funktionen erlauben. Letztere sind somit, wie gefordert, im Austauschfile integriert und werden nun auch zwischen den Systemen ausgetauscht. Der zweite Teil der Forderung nach Funktionen als “first-class-objects” wird in den restlichen, markierten Produktionen erfüllt. Sie unterstützen die Verwendung von Referenzen auf algorithmische Teile zum Zweck der Instanziierung von Entitäten.

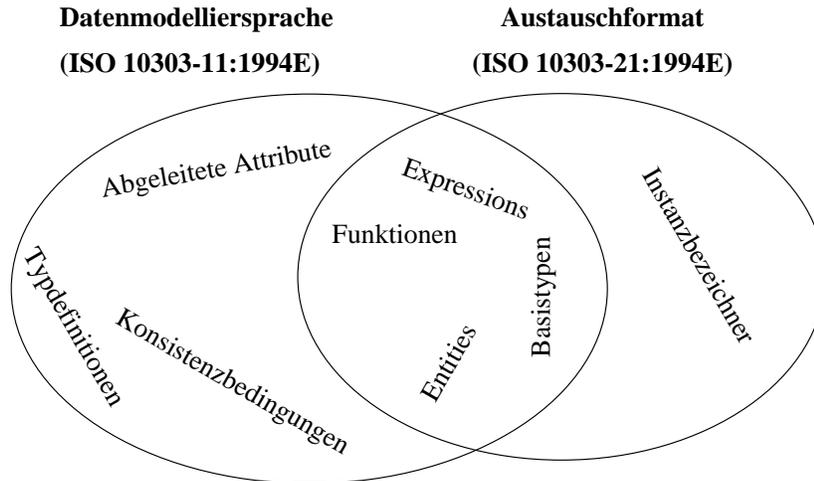


Abbildung 3.5: Datendefinitionssprache/Austauschmodell neu

Die Effekte der Kombination der beiden Grammatikteilen ist in den Abbildungen 3.4 und 3.5 dargestellt. Die existierenden Standards überlappen sich insofern, als dass Entitätsdefinitionen und Basisdatentypen von *EXPRESS* im Austauschformat zur Erzeugung von Instanzen verwendet werden. Neu lassen sich Funktionen und einfache algebraische Ausdrücke, soweit sie parametrische Eigenschaften der Daten darstellen, mit im Austauschformat unterbringen und damit austauschen. Andererseits bleiben Funktionen soweit sie der Modellierung von Konsistenzregeln oder abgeleiteten Attributen dienen, weiterhin ausschließlich im Datenschema und haben keinen Einfluss auf das Austauschformat. Auf diese Art kann die Erweiterung des Austauschformates wie auch der grundlegenden Ideen von [ISO94c] aufwärtskompatibel gestaltet werden. Bereits existierende Schemata und Austauschdaten werden daher nicht invalidiert. Allerdings scheitert der formale Beweis für diese Aussage an theoretischen Grenzen, die durch die verwendete Grammatikklasse gegeben sind [ELP88].

3.3 Semantische Grundlagen

Mit der bisher definierten Grammatik ist nur die Syntax für den Austausch festgehalten. Semantische Festlegungen fehlen hingegen noch und sind Inhalt dieses Kapitels. Dabei wird weniger auf die logische Abgeschlossenheit des Austauschsystems eingegangen, als vielmehr versucht, die wesentlichen Punkte so

darzustellen, dass die Forderungen für den Austausch parametrischer Daten untermauert und eine Grundlage für die später folgenden Erweiterungen gegeben wird.

Die in diesem Kapitel verwendete Schreibweise stammt aus der Logik und findet auch Anwendung in der Typtheorie; eine Einführung zu letzterer findet sich zum Beispiel in [CA96]. Allgemein ist jede Regel so aufgebaut, dass nach dem Namen der Regel eine Anzahl Voraussetzungen (Prämissen) folgen. Diese werden von den daraus resultierenden Konsequenzen durch eine horizontale Linie abgetrennt. Aus der Logik stammt das Zeichen “ \vdash ”. Es wird auch in der Typtheorie verwendet und dient als Ausdruck einer Zusicherung. So bedeutet zum Beispiel die Aussage “ $\Gamma \vdash M : T$ ”, dass (der Bezeichner) M in einer (Austausch-) Umgebung Γ vom Typ T ist.

$$\begin{array}{c}
 \text{(Env } \emptyset) \\
 \hline
 \emptyset \vdash \diamond
 \end{array}
 \qquad
 \begin{array}{c}
 \text{(Type Basis)} \\
 \hline
 \Gamma \vdash \diamond \quad K \in P_1 \cup P_2 \\
 \Gamma \vdash K
 \end{array}$$

Diese beiden Regeln stellen sozusagen die Verankerung des gesamten Typsystems dar. Durch das Fehlen einer Vorbedingung wird die Regel “Env \emptyset ” zu einem Axiom, dessen Inhalt die Forderung darstellt, dass leere Umgebungen (“ \emptyset ”) wohlgeformt sind. (Für die Wohlgeformtheit wird in der Typtheorie üblicherweise das Spezialzeichen “ \diamond ” verwendet.) Unter Umgebung ist im Kontext dieser Arbeit ein Physical File nach [ISO94c], zusammen mit einem Schema zur Definition der austauschbaren Datentypen zu verstehen.

Die Regel “Type Basis” definiert die Basisdatentypen in einer Umgebung. Ausgehend von der zuvor gegebenen Verankerung werden in einem nächsten Schritt die Basisdatentypen eingeführt. Unter P_1 und P_2 sind dabei die Mengen

$$P_1 = \{INTEGER, REAL, STRING, BOOLEAN, LOGICAL\}$$

und

$$P_2 = \{INTEGER^*, REAL^*, STRING^*, BOOLEAN^*, LOGICAL^*\}$$

als Bezeichner für die grundlegendsten Datentypen in *EXPRESS* zu verstehen. P_2 besteht aus den Basisdatentypen, die aus P_1 durch die Erweiterung um ein Element ε (“indeterminate” [ISO94b]) hervorgehen. Dieses Element wird wie der NULL-Wert in relationalen Datenbanken verwendet, also immer dann, wenn eine Information (noch) nicht verfügbar ist. Die Aufteilung in P_1 und P_2 wurde hier vorgenommen, weil eine Datenmodelliersprache durchaus auch ohne die problembeladenen Datentypen aus P_2 definiert werden kann.

Aufbauend auf den vorhergehenden Regeln können die Aggregattypen von *EXPRESS* (List, Array, Bag, Set) definiert werden. Zusätzlich lassen sich auch gleich die Funktionstypen einführen.

$$\begin{array}{c}
\text{(Type Aggr)} \\
\frac{\Gamma \vdash A \quad M, N : \text{INTEGER}^*}{\Gamma \vdash \text{AGGREGATE}(M, N, A)} \\
\text{(Type Func)} \\
\frac{\Gamma \vdash A_1, \dots, A_i, B \quad i \in 0 \dots}{\Gamma \vdash \text{FUNC}(A_1, \dots, A_i) : B}
\end{array}$$

In “Type Aggr” ist der allgemeine Name “AGGREGATE” durch die syntaktische Bildungsregel für spezielle Aggregate in *EXPRESS* zu ersetzen. Also zum Beispiel

$$\begin{array}{c}
\text{(Type Array)} \\
\frac{\Gamma \vdash A \quad M, N : \text{INTEGER}^*}{\Gamma \vdash \text{ARRAY}(M, N, A)}
\end{array}$$

Bei den Termen M und N handelt es sich, wie in der Typtheorie üblich, um Platzhalter für algebraische Ausdrücke, die hier zu einem Wert vom Typ *INTEGER** evaluieren müssen. Die genaue Semantik dieser Ausdrücke ist allerdings erst bei der Definition der Zugriffsoperatoren auf Aggregatelemente relevant. Da Aggregattypen keine neuen Eigenschaften in die Datenmodelliersprache einbringen, wohl aber einen grossen Definitionsaufwand verlangen, werden diese Typen nicht weiter im Detail betrachtet.

Zur Regel “Type Func” ist noch anzuführen, dass die Namen der Parameter wie in der Typlehre üblich nicht relevant sind. Das bedeutet, dass aus einer textuellen Ersetzung der Parameter einer Funktion *keine* neue Funktion hervorgeht¹. Die Kombination von Parametertypen in der Reihenfolge ihres Auftretens und des Resultattyps der Funktion wird als *Funktionssignatur* bezeichnet. Ausserdem sei noch auf die Tatsache hingewiesen, dass im λ -Kalkül nur Funktionen mit einem Parameter bekannt sind. Durch eine verschachtelte Definition solcher Funktionen, die jede als Funktor zu verstehen ist, kann aber die vorliegende Definition nachgebildet werden. Diesen Vorgang nennt sich Currying und ist derart verbreitet, dass hier der Umweg über die einstelligen Funktionen nicht erst gesucht wurde.

$$\begin{array}{c}
\text{(Type SimpleEnt [} l_i \text{ verschieden])} \\
\frac{\Gamma \vdash A_1, \dots, A_m \quad m \in 0 \dots}{\Gamma \vdash \text{ENTITY}(l_1 : A_1, \dots, l_m : A_m) \text{SUB}()}
\end{array}$$

¹im λ -Kalkül entspricht die analoge Substitution einer α -Konversion

(Type ComplEnt [l_i verschieden])

$$\frac{\Gamma \vdash A_1, \dots, A_m, B_1, \dots, B_n \quad m \in 0 \dots, n \in 1 \dots}{\Gamma \vdash \text{ENTITY}(l_1 : A_1, \dots, l_m : A_m) \text{SUB}(B_1, \dots, B_n)}$$

Entitätstypen sind in dem Sinne informell definiert, als dass die Forderung wonach die Basistypen B_1, \dots, B_n in der zweiten Regel Entitätstypen sind, nicht formell aufgestellt wird. Ausserdem wird auch auf die in *EXPRESS* zentrale Idee der Supertype-Expression [ISO94b] nicht formell eingegangen. Das Ziel der Supertype-Expression ist es, spezielle Typkombination von abgeleiteten Entitätstypen zu erlauben, respektive zu verbieten. Dieses Modellbildungsmittel hat für die in diesem Kapitel dargestellten Eigenschaften keine weiteren Auswirkungen.

Wichtig an den Regeln zur Bildung von Entitätstypen ist, dass sie aufzeigen, wie man von einfachen Entitätstypen sukzessive komplexere, abgeleitete Datentypen definieren kann.

Nachdem sich mit den bisher gemachten Definitionen alle momentan interessanten Datentypen bilden lassen, müssen der Austauschumgebung als nächstes Variablen, Referenzen und Instanzen beigefügt werden.

(Intro Var)

$$\frac{\Gamma \vdash A \quad x \notin \text{dom}(\Gamma)}{\Gamma, x : A \vdash \diamond}$$

(Type Ref)

$$\frac{\Gamma \vdash A}{\Gamma \vdash \text{Ref } A}$$

Die Idee eines Referenztyps ist in der Typtheorie relativ neu [AC96], doch scheint dieser Ansatz hier angezeigt, da die Idee der “veränderbaren Zelle einer imperativen, formalen Sprache” [CA96] sicher einfacher weiterzuführen ist, als das mit Elementen aus funktionalen Systemen möglich wäre. Gemäss der vorgegebenen Struktur des Austauschfiles ist zudem klar ersichtlich, dass die Syntax derjenigen von imperativen Programmiersprachen sehr ähnlich ist.

(Val ComplEnt)

$$\frac{\Gamma \vdash A_1, \dots, A_m, B_1, \dots, B_n, N_1 : A_1, \dots, N_m : A_m \quad B = \text{ENTITY}(l_1 : A_1, \dots, l_m : A_m) \text{SUB}(B_1, \dots, B_n), x : B \quad m \in 0 \dots, n \in 1 \dots}{\Gamma, \text{Ref } x : B(N_1, \dots, N_m) \bigoplus_{i=1 \dots n} B_i(\dots) \vdash \diamond}$$

(Val FuncRef)

$$\frac{\Gamma \vdash x : \text{FUNC}(A_1, \dots, A_m) : B \quad m \in 0 \dots}{\Gamma, \text{Ref } x : \text{FUNC}(A_1, \dots, A_m) : B \vdash \diamond}$$

Mit diesen Regeln werden Referenztypen und instanzierbare Werte eingeführt. Es wird gezeigt, wie sich eine Austauschumgebung unter Verwendung von getypten Ausdrücken (“ $N_i : A_i$ ”) mit referenzierbaren Elementen populieren lässt. Die Regel “Val ComplEnt” ist vorerst stark vereinfacht. Sie definiert, wie Instanzen in einer Umgebung erzeugt werden können, wenn man Ausdrücke verwendet, welche zu Typen evaluieren, die gleich den Attributtypen sind. Da vorderhand keine Typkompatibilitätsrelation bekannt ist, müssen die jeweiligen Typen noch exakt übereinstimmen; eine Einschränkung, die im nächsten Kapitel gelockert wird. Im interessanten Fall von Instanzen, deren Typ von mehreren Supertypen abgeleitet ist, werden die einzelnen Typfragmente ähnlich wie zum Beispiel in [ISO94b] separat erzeugt und mit dem Operator “ \oplus ” zusammengesetzt.

Um das Typsystem zu vervollständigen, sind in einem letzten Schritt die Basisoperationen sowie die Verhältnisse zwischen den Typen zu definieren. Die meisten Basisoperationen sind für die Einführung von Funktionstypen nicht sonderlich interessant und werden daher nicht aufgeführt. Einzige Ausnahme ist dabei eine Systemfunktion, die zu jeder Instanz einen eindeutigen Bezeichner (Object Identifier, OID) vom Typ *oid* zurückliefert. Der konkrete Wertebereich dieser Funktion ist nicht wichtig, da hier auf den Werten vom Typ *oid* im Gegensatz zu den Möglichkeiten von einigen relationalen Datenbanksystemen nicht explizit operiert wird. Die Bedingungen an die Werte sind, dass sie

1. jede Instanz einer Umgebung Γ eindeutig kennzeichnen,
2. keine Kollisionen mit Werten anderer Datentypen erzeugen (unterscheidbar zum Zweck des Austausches) und dass sie
3. miteinander vergleichbar sind (also untereinander einer Äquivalenzrelation unterliegen).

Auf syntaktischer Ebene wird mit der Regel “DEFREF” eine solche Kennzeichnung erzeugt.

(Intro OID)

$$\frac{\Gamma \vdash x : \text{Ref } A}{\text{OID}(x) : \text{oid}}$$

Unabhängig von der Definition des Wertevergleichs zwischen Instanzen, muss folgende Beziehung weiterhin gelten:

$$OID(a) = OID(b) \Rightarrow a = b$$

Das bedeutet, *identische* Instanzen sind immer *wertgleich*.

Letztendlich ist das Ziel von typtheoretischen Untersuchungen, Aussagen über fehlerfreie Systeme zu ermöglichen [CA96]. Daher ist das Zusammenspiel der verschiedenen Typen von enormer Wichtigkeit. Um eine Typkompatibilität definieren zu können, muss eine Relation auf den Datentypen der Modelliersprache eingeführt werden. Wie schon früher werden die Aggregationstypen bei dieser Betrachtung ausgeklammert. Sie lassen sich jedoch bei Bedarf recht einfach in die nachfolgende Definition der Ordnungsrelation “ \preceq ” einbinden.

(Eq Refl)

$$\frac{\Gamma \vdash T}{T \preceq T}$$

(Eq Entity)

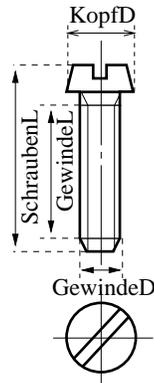
$$\frac{\Gamma \vdash ENTITY(l_1 : A_1, \dots, l_j : A_j)SUB(B_k) \quad j \in 0 \dots, k \in 1 \dots}{\forall 1 \leq v \leq k \bullet ENTITY(l_1 : A_1, \dots, l_j : A_j)SUB(B_k) \preceq B_v}$$

(Eq Func)

$$\frac{\Gamma \vdash FUNC(A_i) : R_1, FUNC(B_i) : R_2 \quad i \in 0 \dots, R_1 \preceq^* R_2, \forall 0 \leq k \leq i \bullet A_k \preceq^* B_k}{FUNC(A_i) : R_1 \preceq FUNC(B_i) : R_2}$$

Mit diesen drei Regeln werden die Grundlagen für Typkompatibilitäten definiert. So wird in der ersten Regel die Bedingung postuliert, dass ein Typ zu sich selbst kompatibel ist. Die Regeln “Eq Entity” und “Eq Func” stellen die Grundlagen für die Kompatibilität zwischen Entitätstypen sowie den neu eingeführten Funktionstypen dar. Wie meistens sind dabei die spezielleren Typen kompatibel zu den generelleren Definitionen. Um die Typkompatibilitätsrelation zu vervollständigen, fehlt einzig noch die Verankerung mit einer Regel für die einfachen Datentypen. Je nachdem wie diese gewählt wird, entsteht als Folge ein streng getyptes System wie zum Beispiel in den Programmiersprachen der Pascal-linie [Wir85, WG92, RW92]. Alternativ kann man statt der strengen Forderung nach einer Äquivalenzrelation auf den Basisdatentypen auch eine Typhierarchie definieren. Eine derartige (partielle) Ordnungsrelation führt zu den schwach getypten Systemen mit impliziten Typumwandlungen (“Coercions”). In beiden Fällen ist jedoch ein konsistentes und abgeschlossenes System definierbar. Im Uebrigen ist die Eigenschaft der Transitivität, die für Ordnungsrelationen notwendig ist, zwar nicht explizit nachgewiesen, doch kann sie leicht aus der rekursiven Bildung der Entitäts- und Funktionstypen abgeleitet werden.

Um zum Schluss auf das eigentliche Ziel, die Unterstützung von parametrischen Daten zurückzukehren, wird nun klar, dass die Parameter einer Austauschumgebung Γ aus genau denjenigen Elementen bestehen, die nur mit der Regel “Intro Var” definiert worden sind. Mangels einer nachfolgenden Instanziierung sind dies somit die freien Variablen innerhalb der Austauschumgebung. Das Informationssystem muss sie erkennen und verwalten. Um diese Aufgabe zu vereinfachen, respektive um die Verwendung von parametrischen Eigenschaften zu Beginn des Datenaustausches anzuzeigen, wird die syntaktische Regel “PARAM_SECTION” verwendet.



```
ENTITY Schraube;
  GewindeL: REAL;
  GewindeD: REAL;
  KopfD:    REAL;
  SchraubenL: REAL;

  WHERE
    GewindeL > 0.0;
    SchraubenL > GewindeL;
    GewindeD < KopfD;
    GewindeD > 0;
END_ENTITY;
```

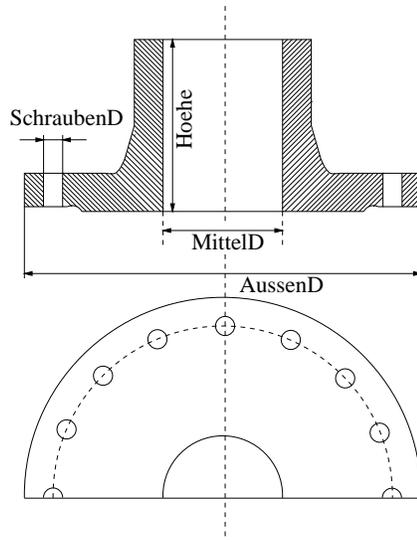
Abbildung 3.6: Schraube, Bild und Modell

3.4 Ausführlicheres Beispiel

Ein etwas aufwendigeres Beispiel ergibt sich aus der Kombination von Schrauben (siehe Abbildung 3.6) mit einem Flansch (schematische Abbildung 3.7 nach [Mas78]). Wenn man die neben den Abbildungen wiedergegebenen Modelle für Flansch und Schraube voraussetzt, so ergibt sich als Beispiel das in Abbildung 3.8 dargestellte Austauschfile.

In der als Beispiel gezeigten Austauschstruktur sind alle bisher neu eingeführten Modell- und Austauschmerkmale vorhanden. Folgende Eigenschaften werden so demonstriert:

- Die Verknüpfung parametrisch definierter Instanzen von Schrauben mit einem Flansch führt zu relativ komplizierten Systemen, die im Beispiel durch eine Bedingung (gleicher Schraubendurchmesser “sd”) miteinander verknüpft sind.
- Funktionale Abhängigkeit der Schraubeninstanz vom Flansch im Attribut



```

ENTITY Flansch;
  --number of screws to
  --mount flange
  SchraubenN: INTEGER;
  SchraubenD: REAL;

  Mitteld:
    FUNCTION(REAL):REAL;
  AussenD: REAL;
  Hoehe: REAL;

  WHERE
    SchraubenN > 0;
    SchraubenN MOD 4 = 0;
    Hoehe > 0.1 * AussenD;
END_ENTITY;

```

Abbildung 3.7: Flansch, Bild und Modell

Kopfdurchmesser. Diese Abhängigkeit stellt sicher, dass der Kopfdurchmesser der Schraube mit dem Bodenteil des Flansches verträglich ist.

- Direkte Verwendung von Werten, wie das bereits in der Originaldefinition des Austauschformates möglich ist.
- Verwendung von Referenzen anstelle von Werten. Neu möglich sind Referenzen auf Funktionen (zB. #40, #30), die als Teil der Austauschstruktur mitübertragen werden.
- Verwendung von algebraischen Ausdrücken (“#20.GewindeL*1.2” respektive “#30(#10)”) anstelle einfacher Werte.
- Freie Parameter (“sd, h, sg”), die erst zu einem späteren Zeitpunkt konkretisiert werden.

3.5 Zusammenfassung

In diesem Kapitel wurde am Beispiel von *EXPRESS* [ISO94b] aufgezeigt, dass eine Erweiterung der Datenmodelliersprache zur Definition und zum Austausch von parametrischen Daten mit wenig Aufwand möglich ist. Dabei sind die Voraussetzungen für die Unterstützung von Parametrik,

1. dass Werte und noch nicht evaluierte Ausdrücke gleich behandelt werden
2. dass Funktionen als referenzierbare Instanzen mit den restlichen Daten gemeinsam ausgetauscht werden.

```

PARAMETER  gl, sd, h: REAL;

DATA
#10= Flansch(12, sd, #40, 32.4, h);
#20= Schraube(gl, sd, #30(#10, sd),#20.GewindeL*1.2);
#30= FUNCTION(f: Flansch; s: Schraube): REAL;
      RETURN MIN(s.KopfD, 0.8*(f.AussenD-f.MittelD)/2);
      END_FUNCTION;
#40= FUNCTION(arg:REAL): REAL;
      RETURN 0.6*arg;
      END_FUNCTION;
END_DATA

```

Abbildung 3.8: Austauschfile für Flansch und Schraube

Damit hat man ein abgeschlossenes, konsistentes System erhalten, das zusätzlich auch zum Austausch, respektive zur Archivierung von Daten mit Constraints im mechanischen Sinne verwendet werden kann. Dies ist unter der Voraussetzung hilfreich, dass der Lösealgorithmus für die Constraints bekannt ist oder mit den Daten selbst ausgetauscht wird.

Kapitel 4

Klassifizierungen nach mehreren Kriterien

In Kapitel 2.3.2 wird gezeigt, zu welchen Problemen die starre Klassifizierung durch eine Klassenhierarchie führt. Das Problem ist deutlich erkennbar, wenn man zum Beispiel versucht, geometrische Objekte zu ordnen. Ein oft verwendetes Kriterium dazu ist der geometrische Typ von Elementen (beispielsweise Kreis, Vektor, Punkt). Alternativ kann aber auch das Transformationsverhalten einzelner geometrischer Objekte zur Klassifizierung herangezogen werden. Beide Vorgehensweisen werden oft gleichzeitig benötigt und daher ist klar, dass im Bereich der Produktdatenmodelle eine Klassifizierung gemäss einem einzelnen Merkmal nicht genügt. Aus diesem Grund soll dieses Problem im Rahmen von STEP aber auch grundsätzlich für die Datenmodellierung untersucht werden.

4.1 Lösungsvorschlag

Offenbar besteht das Problem darin, dass Typhierarchien zu sehr starren Strukturen führen, die zudem durch ein einzelnes Klassifizierungskriterium geprägt sind. Ein vollständiger Verzicht auf eine Typhierarchie andererseits führt zu dynamisch getypten Systemen, die bekannt für ihren Laufzeitaufwand sind. Zudem eignen sich dynamisch getypte Systeme nicht zur Dokumentation von Datenmodellen in Standards, da wesentliche Teile der Modellsemantik nur schwer dargestellt werden können.

Aus diesem Grund muss ein Zwischenweg gesucht werden, der zwar eine Klassifizierung nach Primärmerkmalen ermöglicht, die Verwendung einer flexibleren Ordnung nach weiteren Kriterien aber zulässt. Ein Lösungsansatz sollte aus diesem Grund die Auswahl von statisch definierten, sekundären Merkmalen zur Laufzeit ermöglichen. Damit verbunden sind bedingt gültige Elemente wie Attribute und Konsistenzregeln einzuführen. Dieser Ansatz ermöglicht sowohl eine typsichere Verwendung von Instanzen in der Datenbasis, wie auch eine effiziente Realisierung und eine kontrollierte, dynamische Aenderung von sekundären Kriterien.

4.2 Syntaktische Erweiterungen

Wie schon früher wird als Grundlage für die Erweiterung die Datendefinitionssprache *EXPRESS* [ISO94b] verwendet. Diese lässt sich, wie in der folgenden Syntaxdefinition aufgezeigt, derart erweitern, dass bestehende Modelle nicht invalidiert werden; eine wesentliche Forderung zur Beibehaltung der Kompatibilität.

```

entity      := 'ENTITY' id [subsuper] ';'
              struct_body
              'END_ENTITY'';'.
* struct_body := [dyn_state] {attributes}.
expl_attr   := id {' ',' id' ':'
                 ['OPTIONAL' basetype ''];'.
subsuper    := [supertype] [subtype].
subtype     := 'SUBTYPE' 'OF'
supertype   := 'SUPERTYPE' 'OF' '(' superexpr ')'.
superexpr   := superfact {'AND' | 'ANDOR'} superfact}.
superfact   := id
              | 'ONEOF' '(' superexpr {' ',' superexpr' ')')
              | '(' superexpr ')'.
where_clause := 'WHERE' domain_rule ';' {domain_rule ';'}.
domain_rule := [id ':' ] expression
* dyn_state  := 'TYPE' ':'
*             'ENUMERATION' 'OF'
*             '(' id {' ',' id' ')'.
attributes  := {expl_attr
               ['CASE' 'TYPE' 'OF'
                {state_set ':'
                 {expl_attr
                  [where_clause]}
                'END_CASE' ';' ]}.
* state_set := state {state}.
* state     := '[' id {' ',' id' ']'.
*           [where_clause].

```

Die neuen syntaktischen Elemente finden sich in den mit einem Stern markierten Produktionen. Mit Hilfe der Regel “dyn_state” werden die Eigenschaften aufgezählt, die zur Abgrenzung von sekundären Klassifizierungskriterien dienen. Dabei trägt die Verwendung von Bezeichnern für die Kriterien zur Aussagekraft der Datenmodelle bei. Ausserdem stellt die Grammatik sicher, dass die wichtigsten Informationen innerhalb der Entitätsdeklarationen ausgewiesen werden.

Bei der Definition der Attribute durch die Regel “attributes”, lassen sich neu Attribute definieren, die nur dann gültig sind, wenn spezifizierbare Sekundärkriterien erfüllt sind. Damit wird dem unterschiedlichen Informationsbedarf für Entitäten mit verschiedenen sekundären Klassifizierungen, analog zur Subklassenbildung Rechnung getragen.

```

ENTITY TechPlan;
  TYPE:      ENUMERATION OF (neu, getestet, autorisiert);
  Bez:       Label;
  Titel:     STRING(30);
  Zeichner:  StateInfo;

  CASE TYPE OF
    [getestet]:
      Pruefer: StateInfo;
      bInfo:   Bericht;

    [autorisiert]:
      Autoris: StateInfo;
      fInfo:   Funktion;
  END_CASE;
END_ENTITY;

```

Abbildung 4.1: Modell, Zeichnungsverwaltung

Nimmt man das Beispiel der Dokumentenverwaltung (vergl. Kapitel 2.3.2) wieder auf, so lässt sich mit den in diesem Kapitel gemachten Erweiterungen ein ausdrucksstarkes Modell bilden (siehe Abb.4.1).

4.3 Semantische Grundlagen

Die folgenden semantischen Definitionen machen von der bereits in Kapitel 3.3 verwendeten Notation Gebrauch und erweitern einige der dort gemachten Regeln.

(Intro State)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \text{state } T}$$

Damit die neu eingeführten Zustände sinnvoll zur Modellierung verwendet werden können, ist ein neuer Datentyp notwendig. Er wird mit der Regel “Intro State” definiert. Dabei handelt es sich um einen sogenannten unendlichen Vereinigungstyp (“infinite union type”). Obwohl sich mangels theoretischer Untermauerung die Entwickler dessen nicht bewusst sind, existiert bereits ein ähnlicher Datentyp unter der Bezeichnung “GENERIC” in *EXPRESS*. In einigen andern Programmiersprachen, speziell wenn sie persistente Daten unterstützen, findet sich ein solcher Datentyp ebenfalls [MBCD89, Car85, ABC⁺83]. Diese sind dabei immer als Vereinigung von allen möglichen Datentypen definiert. Hier

wird “stateT” allerdings nur als Vereinigung von spezifizierten Sekundärklassifikationen verstanden. Dennoch zeigen sich die aus der Theorie bekannten Probleme, indem die Grundwerte nicht im voraus als Menge definierbar sind. Die zulässigen Werte des Datentyps “stateT” sind jedoch im konkreten Kontext einer Austauschumgebung eindeutig definierbar. Die Existenz einer wohldefinierten Wertemenge ist ausserdem im Umfeld eines “physical files” [ISO94c] gesichert, da sowohl das Datenschema als auch die ausgetauschten Daten in diesem Rahmen endlich sind. Die Konsequenz von zur Modellierzeit unbestimmten Typmengen ist, dass man entweder die erlaubten Operationen auf derartigen Datentypen einschränken muss, oder dass notwendige Typentests auf einen späteren Zeitpunkt verschoben werden müssen. Der erste Ansatz ist klar nicht brauchbar und so wird im folgenden Teil ein Vorschlag für die zweite Variante aufgezeigt.

Unter der Annahme einer abgeschlossenen Austauschumgebung lässt sich “stateT” in das existierende Typensystem einbetten. Zu diesem Zweck wendet man sich wieder den Entitäten zu und erweitert die früher gemachten Definitionen.

$$\begin{array}{c}
 \text{(Type SimpleEnt [} l_i, s_i \text{ verschieden])} \\
 \\
 \frac{\Gamma \vdash A_1, \dots, A_m, \{s_1 : stateT, \dots, s_k : stateT\} \quad m, k \in 0 \dots}{\Gamma \vdash ENTITY(l_1 : A_1, \dots, l_m : A_m) \{s_1 \dots s_k\} SUB()}
 \end{array}$$

In “Type SimpleEnt” wird wieder vorausgesetzt, dass sowohl die Attributbezeichner, als neu auch die sekundären Merkmalsbezeichner eindeutig sind. Neu lassen sich durch diese Definitionen auch zusätzliche Klassifizierungsmerkmale mit den Entitäten verknüpfen. Derartige Informationen stellen ein zum Datentyp unabhängiges und orthogonales Konzept dar.

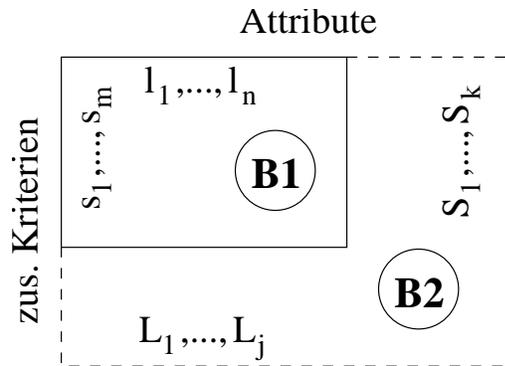
Wiederum werden komplexere, abgeleitete Entitäten rekursiv aus den einfachen Entitätsdefinitionen gebildet. Dazu wird die Regel “Type ComplEnt” wie folgt erweitert:

$$\begin{array}{c}
 \text{(Type ComplEnt)} \\
 \\
 \frac{\Gamma \vdash A_1, \dots, A_m, \{s_k\}, \{s_j\}_1 B_1, \dots, \{s_j\}_n B_n \quad m, j, k \in 0 \dots, n \in 1 \dots}{\Gamma \vdash ENTITY(l_1 : A_1, \dots, l_m : A_m) \{s\} SUB(B_1, \dots, B_n)} \\
 \text{wobei } \{s\} = \{s_k\} \cup \bigcup_{i=1 \dots n} \{s_j\}_i
 \end{array}$$

Zunächst ist zur Form der Regel zu bemerken, dass in Zukunft der Typ “stateT” immer weggelassen wird, wenn die eindeutige Lesbarkeit gegeben ist. Zudem

wird mit der Notation “ $\{s\}$ ” ausdrücklich darauf hingewiesen, dass der Bezeichner “ s ” mengenwertig ist. Ebenfalls neu ist die Notation “ $\{s_k\}B$ ” mit der angedeutet werden soll, dass der Typ B mit einer Menge von zusätzlichen Merkmalen ($\{s_1 : stateT, \dots, s_k : stateT\} k \in 0 \dots$) verknüpft sein kann. Damit handelt es sich bei der Regel “Type ComplEnt” um eine der wichtigeren Definitionen dieses Unterkapitels.

Da im Rahmen der Definition von Entitäten alle gültigen Sekundärmerkmale aufgezählt werden, basieren die Werte von “stateT” zu jedem Zeitpunkt auf einer endlichen Menge. Aus diesem Grund kann von der Mengenvereinigung bei der Ableitung von Typen Gebrauch gemacht werden. Daraus ergibt sich, dass sekundäre Klassifikationsmerkmale über die Datenhierarchie vererbbar sind. Ebenfalls wie bei der Spezialisierung mittels Unterklassen lassen sich die Kriterien in den Subtypen erweitern. Dieser Vorgang ist in Abbildung 4.2 für einen einfachen Fall graphisch dargestellt.



$$B1 = ENTITY(l_1, \dots, l_n)\{s_1, \dots, s_m\}SUB()$$

$$B2 = ENTITY(L_1, \dots, L_j)\{S_1, \dots, S_k\}SUB(B1)$$

Abbildung 4.2: Vererbung von Zuständen

Ausgehend von einem Entitätstyp, bestehend aus den orthogonalen Konzepten von Attributen und den Merkmalen s_1, \dots, s_n , wird ein Subtyp (“ $B2$ ”) gebildet. Dieser erbt sämtliche Attribute und Merkmale von seinem Supertyp (“ $B1$ ”) und erweitert sie gleichzeitig. Daraus ergeben sich die Sekundärmerkmale S_1, \dots, S_k und die Attribute L_1, \dots, L_j . An der Orthogonalität von zusätzlichen Klassifikationskriterien und Attributen ändert sich aber, wie in der Abbildung angedeutet, nichts. Mit dem früher aufgeführten Beispiel könnten $B1$ und $B2$ Vektoren bezüglich eines globalen respektive lokalen Koordinatensystems sein. Ein Sekundärkriterium ist dann zum Beispiel die Transformierbarkeit.

Nachdem die Definition der Typenbildung und der Vererbung von Zuständen erfolgte, muss in einem nächsten Schritt wieder die Erzeugung von Instanzen untersucht werden. Zu diesem Zweck benötigt man noch eine Hilfsfunktion. Diese testet, ob ein Attribut einer Instanz mit gegebenem Typ in einem spezifischen Zustand sichtbar ist.

(visible)

$$\begin{array}{c}
\Gamma \vdash \{s_j\}_1 B_1, \dots, \{s_j\}_n B_n, A_1, \dots, A_m \\
B = ENTITY(l_1 : A_1, \dots, l_m : A_m) \{s_k\} SUB(B_1, \dots, B_n) \\
n, m, j, k \in 0 \dots \quad \{S\} \subseteq \{s_k\} \cup \bigcup_{i=1 \dots n} \{s_j\}_i \quad q \in 1 \dots m \\
\hline
\Gamma \vdash visible(B, l_q, \{S\}) : \text{BOOLEAN}
\end{array}$$

Das bedeutet, dass in jeder wohlgeformten Austauschumgebung eine Funktion *visible* existiert. Mit dem früher geäußerten Verständnis von Typmengen ist die Funktion auf Kombinationen von Attributen, Entitäten und Merkmalsbezeichnern definiert. Die Existenz der Attribut- und Entitätsmenge ist natürlich ebenfalls durch die endliche Grösse des Datenmodells gegeben. Um “*visible*” verwenden zu können, muss in einem zweiten Schritt die operationelle Semantik definiert werden; erst sie spezifiziert ein “Verhalten” für die soeben eingeführte Funktion.

$$visible(B, l, \{S\}) : \text{BOOLEAN} = \begin{cases} \text{TRUE} & \text{falls das Attribut } l_i \text{ von} \\ & B \text{ mit keinem Zustand} \\ & \text{assoziiert oder in} \\ & \text{mindestens einem der} \\ & \text{durch } \{S\} \text{ gegebenen} \\ & \text{Zustände sichtbar ist} \\ \text{FALSE} & \text{sonst} \end{cases}$$

Im übrigen ist darauf hinzuweisen, dass die Funktion “*visible*” nur im formalen Teil dieser Arbeit Verwendung findet. Sie dient dazu, nachfolgende Regeln einfacher formulieren zu können und ist daher in den Datenmodellen nicht verfügbar.

Mit Hilfe der Sichtbarkeitsfunktion “*visible*” kann die früher gemachte Definition gültiger Instanzierungen nun auch für Objekte mit zusätzlichen Klassifizierungen erfolgen.

(Val StateSimpleEnt)

$$\begin{array}{c}
\Gamma \vdash A_1, \dots, A_m, T_1, \dots, T_p, N_1 : T_1, \dots, N_p : T_p \\
B = ENTITY(l_1 : A_1, \dots, l_m : A_m) \{s_k\} SUB(), x : B \\
m, k \in 0 \dots, p \leq m
\end{array}$$

$$\begin{array}{c}
s \subseteq \{s_k\} \quad V = \langle A_j, j \in 0 \dots m \mid visible(B, l_j, \{s\}) \rangle \\
\forall A_j \in V \bullet T_j \preceq^* A_j
\end{array}$$

$$\Gamma, Ref \ x : B(s, N_1, \dots, N_m) \vdash \diamond$$

Intuitiv ist klar, dass eine Instanz durch die Angabe von Klassifizierungsmerkmalen sowie der Werte sämtlicher gültiger Attribute eindeutig instanzierbar ist. Dabei sollen die Attribute in der durch ihre Definition gegebenen Reihenfolge spezifiziert werden. Aus diesem Grund wird in der Definition die Sequenz V als Hilfselement eingeführt.

Da diese neuen Regeln lediglich Erweiterungen der früher gemachten Definitionen darstellen, werden die kompakteren Schreibweisen für Instanzen ohne Sekundärkriterien beibehalten. Die kürzeren Schreibweisen (zB. “Val ComplEnt”), der hier gemachten Definitionen (zB. “Val StateSimpleEnt”), sind im übrigen nur Kurzversionen für den Fall $s = \emptyset$.

Komplexe Instanzen, wie sie in [ISO94b] und [ISO94c] definiert sind, lassen sich unter Berücksichtigung von Klassifizierungskriterien rekursiv aus den einfachen Instanzen aufbauen.

(Val StateComplEnt)

$$\begin{array}{l} \Gamma \vdash A_1, \dots, A_m, B_1, \dots, B_n, T_1, \dots, T_p, N_1 : T_1, \dots, N_p : T_p \\ B = ENTITY(l_1 : A_1, \dots, l_m : A_m) \{s_k\} SUB(B_1, \dots, B_n), x : B \\ m, k \in 0 \dots, p \leq m, n \in 1 \dots \end{array}$$

$$\begin{array}{l} s \subseteq \{s_k\} \quad V = \langle A_j, j \in 0 \dots m \mid visible(B, l_i, \{s\}) \rangle \\ \forall A_j \in V \bullet T_j \preceq^* A_j \end{array}$$

$$\Gamma, Ref \ x : B(s, N_1, \dots, N_p) \bigoplus_{i=1 \dots n} B_i(\dots) \vdash \diamond$$

Dabei werden einzelne, sogenannte partielle Instanzen wie mit der Regel “Val StateSimpleEnt” erzeugt und mit Hilfe des Operators “ \oplus ” zu einer komplexen Instanz verknüpft. Dieser Operator wird in [ISO94b] mit “ $||$ ” definiert, wohingegen er in [ISO94c] durch die sequentielle Aufzählung der partiellen Instanzen nur implizit vorhanden ist.

Um in den Datenmodellen grösstmögliche Typsicherheit einbauen zu können, benötigt der Modellierer eine Funktion, die ihm erlaubt, Instanzen auf die Gültigkeit von Klassifizierungsmerkmalen zu testen. In Abhängigkeit vom Resultat dieser Auswertung können die bedingt gültigen Attribute auf sichere Art und Weise dereferenziert werden. In einigen andern Programmiersprachen existiert ein ähnliches Hilfsmittel unter dem Begriff “type test” für erweiterbare Typen [RW92, Mös94, Ode89] respektive Diskriminator für Variantentypen [Wir85]. In beiden Fällen handelt es sich um Konzepte, die dem hier gemachten Ansatz ähnlich sind, aber die existierenden Systeme nicht mit einem orthogonalen Konzept ergänzen.

(StateTest)

$$\frac{\Gamma \vdash x : ENTITY(l_1 : A_1, \dots, l_m : A_m) \{s_k\} SUB(B_1, \dots, B_n), \{s_j\}_1 B_1, \dots, \{s_j\}_n B_n \quad m, n, k \in 0 \dots, s \subseteq \{s_k\} \cup \bigcup_{i=1 \dots n} \{s_j\}_i}{\Gamma \vdash x. TYPE : SET OF stateT}$$

Damit das angestrebte Ziel, einer typsicheren Modellierung erreicht werden kann, muss die operationelle Semantik derart ausgelegt werden, dass “StateTest” für sämtliche gültigen Instanzen einer Austauschumgebung definiert ist. Das führt dann zu

$$x. TYPE : SET OF stateT = \begin{cases} \{s\} & \text{falls } x \text{ mit einer in der} \\ & \text{Umgebung } \Gamma \text{ gültigen} \\ & \text{Instanz populiert ist} \\ & \text{und die} \\ & \text{Entitätsdefinition} \\ & \text{verschiedene Zustände} \\ & \text{kennt} \\ \{\} & \text{andernfalls} \end{cases}$$

Der Rückgabewert der Funktion besteht demnach in der Menge der zutreffenden Merkmale respektive der leeren Menge.

Ebenfalls benötigt wird ein Weg, den gerade aktuellen Zustand zu wechseln. Die entsprechende Definition findet sich in der Regel “StateChange” deren operationelle Semantik besagt, dass sich Instanzen *nach* Ausführung von “StateChange” im neu gesetzten Zustand befinden. So soll für “StateChange” gelten: $x. TYPE := s \Rightarrow x. TYPE = s$.

(StateChange)

$$\frac{\Gamma \vdash \{s_j\}_1 B_1, \dots, \{s_j\}_k B_k, x : ENTITY(l_1 : A_1, \dots, l_n : A_n) \{s_m\} SUB(B_1, \dots, B_k) \quad n, k, m \in 0 \dots \quad s \subseteq \{s_m\} \cup \bigcup_{i=1 \dots n} \{s_j\}_i}{\Gamma \vdash x. TYPE := s}$$

In Analogie zur Regel “StateTest” müssen Operatoren zur typsicheren Behandlung von Datentypen zu Verfügung gestellt werden.

(isType)

$$\begin{array}{c}
\Gamma \vdash A_1, \dots, A_m, B_1, \dots, B_n, T, x : B, \\
B = ENTITY(l_1 : A_1, \dots, l_m : A_m)\{s_k\}SUB(B_1, \dots, B_n) \\
m, k, n \in 0 \dots T \preceq^* B \vee B \preceq^* T \\
\hline
\Gamma \vdash isType(x, T) : BOOLEAN
\end{array}$$

Diese Funktion erlaubt dem Modellierer zu testen, ob sich hinter einer Variablen eine Instanz von einem gegebenen Typ verbirgt. Dabei ist klar, dass der Test für den Fall $B \preceq^* T$ statisch auswertbar ist. In *EXPRESS* existiert bereits eine ähnliche Funktion unter dem Namen “*TypeOf*”. Sie ist definiert als

$$TypeOf(V : GENERIC) : SET OF STRING$$

und kann somit die Eigenschaften von “isType” emulieren. Allerdings sind Typentests auf der Basis von Zeichenkettenvergleichen aufwendig und sehr fehleranfällig.

(TypeGuard)

$$\begin{array}{c}
\Gamma \vdash A_1, \dots, A_m, B_1, \dots, B_n, T, x : B \\
B = ENTITY(A_1, \dots, A_m)\{s_k\}SUB(B_1, \dots, B_n) \\
m, k, n \in 0 \dots T \preceq^* B \vee B \preceq^* T \\
\hline
\Gamma \vdash x \setminus T : T
\end{array}$$

Die Funktion “TypeGuard” existiert in [RW92, Mös94, Ode89] und dient dazu, sicherzustellen, dass eine Variable (“*x*”) einen bestimmten dynamischen Typ (“*T*”) einer Instanz enthält. Falls das nicht der Fall ist, löst die Funktion bei Programmiersprachen im Laufzeitsystem einen Fehler aus. Im Rahmen des Datenaustausches muss das Informationssystem in geeigneter Masse auf diesen Fehlerzustand reagieren. Auch hier ist klar, dass sich für den Fall $B \preceq^* T$ der Typentest erübrigt, da die Bedingung trivialerweise erfüllt ist. In *EXPRESS* existiert ein ähnlicher Operator unter der Bezeichnung “Group Reference Operator”. Diese Operation ist trotz fehlender formaler Definition besser in die Modellersprache eingebettet als die zuvor gezeigte Funktion “TypeOf”.

Da der Typeguard eine der fundamentalen Funktionen in einer objektorientierten Modellersprachen ist, wird seine Laufzeiteigenschaft nachfolgend auf eine formale Basis gestellt.

(Dyn TypeGuard)

$$\begin{array}{l}
\Gamma \vdash B_1, \dots, B_n, A_1, \dots, A_m, A, T, x : Ref A, \\
B = ENTITY(l_1 : A_1, \dots, l_m : A_m) \{s_k\} SUB(B_1, \dots, B_n), \\
Ref x : B(\dots) \bigoplus_{i=1 \dots n} B_i(\dots) \quad m, k, n \in 0, \dots, B \preceq^* T \preceq^* A \\
\hline
\Gamma \vdash Ref x \setminus T : T
\end{array}$$

Die Definition der Laufzeitsemantik beschränkt sich auf den interessanten Fall, in dem das Gelingen nicht trivialerweise garantiert werden kann. Vorausgesetzt wird, dass sich Typen B , T und A finden lassen, sodass B ein zu A und T kompatibler Subtyp ist. Als weitere Voraussetzung wird die Existenz einer Instanz vom dynamischen Typ B angenommen, die über eine Variable vom Typ A referenziert werden kann. Mit Hilfe des Typeguards kann dann eine neue Sicht vom dynamischen Typ T auf die Instanz B erzeugt werden. Diese Regel deckt den allgemeinsten Fall ab, in dem drei Objekttypen zur Anwendung kommen. Indem man $T \equiv A$ respektive $T \equiv B$ wählt, erhält man die üblichen Spezialfälle. (Allerdings stellt der Fall $T \equiv A$ eine Nulloperation dar, da er trivialerweise erfüllt ist.)

Im Unterschied zur Definition in *EXPRESS*, liefert die hier gemachte Definition des Typeguards eine besser verwendbare Funktion. So erreicht man durch die Wahl der Vorbedingung, dass die in der Operation verwendeten Typen in einem Verhältnis zueinander stehen müssen. Aus diesem Grund ist die Funktion immer auswertbar und liefert ein weiterverwendbares Ergebnis. (So führt $x \setminus T.attr$ einen Typencheck durch, liefert eine neue Sicht auf x , falls der Text positiv verlief, und spricht schlussendlich das Attribut $attr$ an.) In *EXPRESS* sind die eingesetzten Typen frei wählbar, ohne in einer Beziehung stehen zu müssen. Dadurch sind auch Situationen denkbar, die keine Auswertung erlauben. In diesen Fällen liefert *EXPRESS* den Wert *indeterminate* ('?') zurück. Damit wird die Definition zwar konsistent, doch ist eine sequentielle Verwendung mehrerer Operatoren nicht mehr möglich.

Mit diesen beiden Funktionen stehen dem Datenmodellierer zwei ausdrucksstarke Mittel zu Verfügung, um den dynamischen Typ einer Instanz zu garantieren (TypeGuard), respektive um eine Instanz auf ihren dynamischen Typ zu testen (isType).

Eine weitere, wichtige Funktion im Zusammenhang mit den neu eingeführten Zustandsinformationen ist die Dereferenzierung. Mit dem Ziel, bedingt gültige Attribute zu unterstützen, lässt sich die Semantik der Dereferenzierung verbal folgendermassen formulieren:

Attribute einer Instanz lassen sich genau dann dereferenzieren, wenn entweder das Attribut unabhängig vom Zustand der Instanz dereferenzierbar ist, oder aber wenn die Instanz Sekundärkriterien erfüllt, in denen bedingt sichtbare Attribute gültig sind.

Die formale Version dieser Aussage ist gegeben durch den Ausdruck:

(Deref)

$$\begin{array}{c}
 \Gamma \vdash A_1, \dots, A_m, \{s_j\}_1 B_1, \dots, \{s_j\}_n B_n, \{s\}, x : B, \\
 B = \text{ENTITY}(l_1 : A_1, \dots, l_m : A_m) \{s_k\} \text{SUB}(B_1, \dots, B_n) \\
 m, n, k \in 0 \dots, s \subseteq \{s_k\} \cup \bigcup_{i=1 \dots n} \{s_j\}_i \\
 \exists T, i \bullet \Gamma \vdash T \wedge B \preceq^* T \wedge \text{visible}(T, l_i, \{s\}) \\
 \hline
 \Gamma \vdash M.l_i : A_i
 \end{array}$$

Da die meisten objektorientierten Systeme keine Typqualifikation bei der Dereferenzierung erzwingen, solange als die geerbten Attribute eindeutig sind, erweist sich die formale Spezifikation der Semantik dieses Operators als relativ aufwendig.

Dennoch zeigt die Regel ‘‘Deref’’ die Moglichkeiten fur eine effiziente Implementierung deutlich. So lasst sich die Definition fur alle Objekte deren Attribute unabhangig von Merkmalen sind, zur Regel ‘‘Deref*’’ vereinfachen.

(Deref*)

$$\begin{array}{c}
 \Gamma \vdash A_1, \dots, A_m, B_1, \dots, B_n \quad m, n \in 0 \dots, \\
 B = \text{ENTITY}(l_1 : A_1, \dots, l_m : A_m) \text{SUB}(B_n), x : B \\
 \exists T, i \bullet \Gamma \vdash T \wedge B \preceq^* T \wedge \text{visible}(T, l_i, \{\}) \\
 \hline
 \Gamma \vdash M.l_i : A_i
 \end{array}$$

Uebertragt man die Definition auf die operationelle Semantik, so zeigt es sich, dass die Vorbedingung $\text{visible}(T, l_i, \{\})$ statisch voll auswertbar ist. Das bedeutet aber, dass im Gegensatz zu den Vorschlagen in [SNS94] mit einem vollig freien Typsystem, die Typ- respektive Merkmaltests in vielen Fallen eingespart werden konnen. Damit reduziert sich der Zusatzaufwand durch Laufzeittests auf die Falle, in denen Attribute dereferenziert werden, die bedingt sichtbar sind. Die Mehrheit der Instanzen eines Produktdatensystems erfahren also keinen Mehraufwand und konnen effizient behandelt werden.

Leider haben die Entwickler der Datenmodelliersprache einen Algorithmus fur den Wertevergleich von Instanzen definiert [ISO94b]. Er muss daher fur eine sinnvolle Berucksichtigung von zustandsabhangigen Attributen angepasst werden. Dies ist nicht sonderlich schwierig und bedingt einzig eine Beschrankung der Wertevergleiche auf sichtbare Attribute. Aus diesem Grund muss der Vergleichsalgorithmus um die fett markierten Teile erweitert werden (s. Abb. 4.3).

Der *ValueEqual(l, r)* Algorithmus:

- a) falls $oid(l) = oid(r)$ evaluiere zu *TRUE*
- b) initialisieren einer leeren Liste *plist*
- c) Resultat von *DeepEqual(l, r, plist)* zurückgeben

Der *DeepEqual(l, r, plist)* Algorithmus:

- a) wenn sowohl l, als auch r zu "indeterminate" ('?') evaluieren, so ist das Resultat des Vergleichs *FALSE*
- b) falls $TYPEOF(l) \neq TYPEOF(r)$ so wird *FALSE* zurückgegeben
- c) **falls die Instanzen bedingt sichtbare Attribute haben und in verschiedenen Zuständen sind wird *FALSE* zurückgegeben**
- d) falls l und r keine Instanzen sind, so ist das Resultat des einfachen Wertvergleichs zurückzugeben
- e) falls $oid(l) = oid(r)$ resultiert der Algorithmus in *TRUE*
- f) falls das Paar (l, r) schon einmal getestet wurde und daher in *plist* existiert, wird *TRUE* zurückgegeben
- g) andernfalls
 1. füge das Paar (l, r) der Liste *plist* hinzu und
 2. rufe den *DeepEqual* Algorithmus für alle **sichtbaren** Attribute auf

Abbildung 4.3: Wertevergleichsalgorithmus

Gemäss diesem Algorithmus sind zwei Objekte also genau dann wertegleich, wenn bei der Ausführung der beschriebenen Vergleichen *TRUE* resultiert.

4.4 Evolution

Im Gebiet der Datenbanken wird der Begriff "Evolution" in einigen verschiedenen Bedeutungen verwendet. Die häufigsten sind:

- Erzeugen und Löschen von Instanzen
- Aendern des Typs einer im Informationssystem existierenden Instanz
- Aendern des Datenschemas eines Informationssystems

Dabei spielt der erste Punkt in Rahmen dieser Arbeit, die sich nur mit der Datenmodellierung befasst, keine Rolle, sondern ist bei der Verwendung des

STEP-Standards [ISO94a] in den Applikationen unter Verwendung der Datenzugriffsdefinitionen [Pri94] zu behandeln.

Der zweite Punkt wird oft mit dem Begriff “Objektevolution” bezeichnet, während der dritte Punkt als “Schemaevolution” immer noch Gegenstand heutiger Forschung darstellt. In den nachfolgenden Unterkapiteln wird aufgezeigt, wie der Vorschlag, verschiedene Klassifikationen einzuführen, bei der Behandlung von Fragen der Evolution zu helfen vermag.

4.4.1 Objektevolution

In den traditionellen Programmiersprachen ist die Objektevolution ein bekanntes und prinzipiell gelöstes Problem. Eine Lösung ist jedoch nur möglich, da existierende Programmiersprachen spezielle Eigenschaften aufweisen. So sind Daten in Applikationen kurzlebig, womit sich die Frage der Schemaevolution kaum stellt. Zudem werden Daten meist auch nicht gleichzeitig von mehreren Anwendungen benutzt und verändert. Damit entschärfen sich die Probleme der Objektevolution. Als Folge dieser Eigenschaften ist die Kontrolle des Verhaltens von Informationseinheiten während der Ausführungszeit von Anwendungen gesichert.

Sobald aber persistente Daten oder Datensharing eingesetzt werden, zeigen sich noch ungelöste Probleme. Sie treten dann auf, wenn existierende Instanzen ihre Merkmale ändern, ohne dass sie deswegen einem andern Datentyp zugeordnet werden dürfen. Eine Umklassifizierung existierender Daten, das bedeutet eine Neuuzuordnung von Datentypen kann grundsätzlich auf zwei Arten geschehen. Entweder wird der gesamte Datenbestand reorganisiert (“greedy algorithm”), was aber nicht nur ein grosser Aufwand ist, sondern auch das Datensystem während einiger Zeit blockiert. Als Alternative kann eine Umwandlung immer dann geschehen, wenn auf einzelne Datensätze zurückgegriffen wird (“lazy conversion”).

Meist ist in der Produktdatenindustrie jedoch keiner der beiden Wege gangbar, existierende Instanzen dürfen aus rechtlichen Gründen nicht umkopiert und dabei mit einem neuen Objektidentifikator versehen werden. Dadurch wäre die Beweisfähigkeit der Daten nicht mehr gegeben, und zudem verlangen objektorientierte Datensysteme mit ihren stark vernetzten Instanzen eine besonders aufwendige Datenreorganisation.

Als Konsequenz muss der Modellierer einfache und sichere Mittel erhalten, um beide Szenarien unterstützen und dokumentieren zu können, ohne dass im späteren Gebrauch aufwendige Anpassungen der Verwaltungsinformationen über die Daten notwendig werden.

Trotz der Bedeutung der Objektevolution wurde dieses Problem gegenüber den Fragen der Schemaevolution vernachlässigt. So ist es nicht verwunderlich, dass existierende Lösungsansätze [SLR⁺92, KBC⁺89, BMO⁺89] zu einem relativ hohen Systemaufwand führen, weil zu oft dynamische Typentests durchgeführt werden müssen. Das bedeutet aber auch, dass die Evolution schlecht dokumentiert und kaum kontrolliert werden kann, da die verwendeten Mecha-

nismen zur Typänderung von den Datendefinitionen getrennt in den Anwendungen zu finden sind. Dadurch wird die Dichotomie in eine Strukturdefinition innerhalb eines Schemas und in einen semantischen Teil in den Anwendungen andererseits [VH91] weiter unterstützt.

Eine erfolgsversprechende Lösung findet sich bei der Suche nach voneinander unabhängigen Konzepten, wie das beispielsweise in [NSWW96] mit Typen und Rollen vorgeschlagen wird. Die vorgeschlagenen, bedingt sichtbaren Attribute erlauben eine kontrollierte Objektevolution, indem die Primärklassifikation beibehalten wird, und eine gewisse Evolution über die zusätzlichen Klassifizierungskriterien erlaubt ist. Dieser Ansatz erlaubt zudem einen effizienten Zugriff auf Relationen, die bezüglich der Evolution invariant sind.

4.4.2 Schemaevolution

Fragen der Schemaevolution wurden bereits im Zusammenhang mit relationalen Datenbanken untersucht [Rod93]. Das Problem ist bei relationalen Datenbanken insofern einfacher, als Daten durch ihre Werte bestimmt sind. Dadurch stellen sich die Probleme der Referenzen über Objektidentifikatoren im relationalen Datenmodell nicht. Das wiederum ermöglicht eine Reorganisation der Werte in einem Datenbanksystem bei gleichzeitiger Anpassung der Anwendungsprogramme.

Heute hat sich die Erkenntnis durchgesetzt, dass nicht alle erdenklichen Schemaänderungen vom Datenbanksystem unterstützt werden müssen. Dies zumal es Änderungen gibt, die mehrere semantische Deutungen haben können. So wird in den neuesten Untersuchungen meist eine Taxonomie der möglichen Evolutionsschritte definiert [PÖ95, PS87, BKKK87]. Darauf aufbauend können dann die von einem System unterstützten Schemamodifikationen definiert und mit einer Semantik versehen werden.

In diesem Zusammenhang kann der Vorschlag von bedingt gültigen Attributen, respektive des objektorientierten Ansatzes in den folgenden Fällen behilflich sein:

- Erweiterung einer existierenden Typdefinition
- Reduktion einer existierenden Typdefinition

Objektorientierung entstand, um die Spezialisierung von Datentypen in dem Sinn zu vereinfachen, dass bestehende allgemeine Funktionalitäten weiterhin benutzbar blieben. Aus diesem Grund ist es nicht erstaunlich, dass erweiterte Daten wie in Tabelle 4.1 gezeigt mit Daten aus einer alten Umgebung gut koexistieren können. Mit Hilfe der Eigenschaft des Polymorphismus ist es möglich, erweiterte Instanzen gemäss einem neueren Datenmodell weiterhin in der Umgebung zu verwenden, die sich auf ein älteres Datenmodell abstützt [Os88].

Für die Industrie relevanter ist jedoch der Fall, in dem die alten Daten mit einem neuen, erweiterten Datenmodell zusammenarbeiten müssen. Dieser

Quellumgebung	Zielumgebung	
	alte	neue
alte	keine Aenderung nötig	anpassen der alten Definitionen, Sekundärmerkmale!
neue	Daten brauchbar infolge Polymorphismus	keine Aenderung nötig

Tabelle 4.1: Typerweiterung

Fall lässt sich mit Hilfe von sekundären Klassifizierungskriterien recht elegant lösen, indem man ein Merkmal einführt, das die Version der Entitätsdefinition beschreibt. Die operationellen Teile bedienen sich dieser Merkmale um sicherzustellen, dass nur gültige Zugriffe auf die verschiedenen Attribute vorkommen. Wie bereits dargestellt, beschränkt sich der Mehraufwand bei diesem Ansatz auf diejenigen Attribute, die versionsabhängig sind. Alle ändern Attribute und Typen müssen weder angepasst werden, noch führt deren Verwendung zu einem Mehraufwand.

Quellumgebung	Zielumgebung	
	alte	neue
alte	keine Aenderung nötig	keine Anpassung, falls Evolution über Sekundärkriterien erfolgt
neue	alte Quellen anpassen	keine Aenderung nötig

Tabelle 4.2: Typreduktion

Der zweite interessante Fall, in dem aus einer Objektdefinition Attribute entfernt werden (siehe Tab 4.2), zeigt die Verwendbarkeit des Zustandkonzepts besonders schön. Eine Typenänderung ist nach traditionellen Ansätzen nur mit einer grösseren Reorganisation der bestehende Daten möglich. Gleichzeitig muss eine Oberklasse sowie eine neue Unterklasse bestehend aus den reduzierten Attributen gebildet werden. Derartige Aenderungen der Typhierarchie verlangen meist auch grössere Anpassungen in den Anwendungen. Modelliert man aber die Aenderung mittels zusätzlichen Klassifikationskriterien, so sind sämtliche alten Instanzen in der neuen Umgebung ohne Einschränkung verwendbar. Die alten Instanzen beinhalten zwar gegenüber den neuen Definitionen ein zusätzliches Attribut, doch ist dieser Fall analog zur Subtypenbildung und Verwendung des Polymorphismus zur gemeinsamen Verwendung erweiterter Typen. Der umgekehrte Fall, in dem Instanzen nach dem reduzierten Datenmodell in der alten Umgebung verwendet werden, ist in der Praxis kaum relevant, doch lässt sich auch dieser Fall durch die Verwendung von bedingt gültigen Attributen und

entsprechenden Tests im alten Code absichern.

Viele der weiteren interessanten Fälle einer Schemaentwicklung lassen sich als Kombination dieser Elementaroperationen darstellen, oder dann in analoger Weise, durch das Anpassen der alten und neuen Definitionen lösen.

Relevant dabei ist, dass die Semantik der Datentypen möglichst im Datenbankschema untergebracht werden kann.

Das erspart die Anpassung und Recompileation derjenigen Anwendungen, die auf den Daten operieren. Als weiterer Vorteil erweist sich die Verwendung von zusätzlichen Klassifizierungskriterien zur Kontrolle der Typenvielfalt. Mit Hilfe dieses Konzepts ist der Modellierer nicht mehr gezwungen, artifizielle Subtypen einzuführen, um die Schemaevolution abbilden zu können. Besonders bei *EXPRESS* führen tiefe Klassenhierarchien aufgrund der Supertypexpression zu kaum beherrschbaren Kombinationen der Typen. Dieses Problem kann bereits bei relativ wenigen Entitätsdefinitionen nur noch mit maschineller Hilfe kontrolliert werden.

4.5 Ausführlichere Beispiele

Die Verwendung der in diesem Kapitel vorgestellten Konzepte wird im folgenden an konkreten Beispielen verdeutlicht. Zu diesem Zweck wird das bereits eingeführte Datenmodell mit den Objekten “Flansch” und “Schraube” ausgebaut.

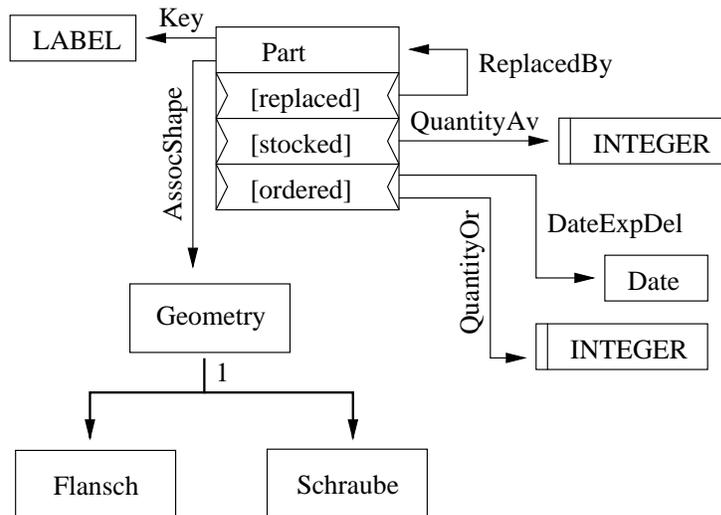


Abbildung 4.4: Teilemodell, *EXPRESS-G*

In einem ersten Schritt wird für die beiden unabhängigen Entitäten ein gemeinsamer Supertyp geschaffen. Diesen verwendet das neue Objekt “Part” um zu zeigen, wie sich die Typevolution mit Hilfe von Zuständen darstellen lässt. Abbildung 4.4 stellt eine graphische Version eines Datenmodells für den untersuchten Fall dar. Die graphische Darstellung basiert auf *EXPRESS-G* [ISO94b], wurde aber um Darstellungsmittel für die neuen Konzepte erweitert. Zusätzliche Klassifizierungsmerkmale werden in der Graphik innerhalb des Symbols für Entitäten gezeichnet. Relationen die von einem Zustandsbezeichner ausgehen markieren die Verknüpfung als bedingt gültig.

Mit dieser Information lässt sich das Datenschema in Abbildung 4.4 so interpretieren, dass das Objekt “Part” über drei Merkmale (“replaced”, “ordered” und “stocked”) als sekundäre Klassifikationsmittel verfügt. Unabhängig davon existieren die Attribute “Key” und “AssocShape”. (Letztere bildet die Verbindung von einem “Part” zu seiner Geometrieinformation.) Weiter beinhaltet die Entität “Part” bedingt sichtbare Attribute, namentlich die Information wieviele Teile eines Typs an Lager sind (“QuantityAv”), wieviele Teile bestellt (“QuantityOr”), auf wann diese erwartet werden (“DateExpDel”), sowie ein Ersatzteil (“ReplacedBy”). Jede dieser Relationen ist allerdings davon abhängig welche Sekundärkriterien zutreffen. Die entsprechende Textversion des Datenmodells findet sich in Figur 4.5.

```

ENTITY Geometry
  SUPERTYPE OF ONEOF(Flansch, Schraube);
END_ENTITY;

ENTITY Part;
  TYPE = SET OF(stocked, ordered, replaced);
  Key: LABEL;
  AssocShape: Geometry;
  -- state oriented definitions
CASE TYPE OF
  [stocked]:
    QuantityAv: INTEGER;
  [ordered]:
    QuantityOr: INTEGER;
    DateExpDel: Date;
  [replaced]:
    ReplacedBy: Part;
    WHERE ReplacedBy :<>: SELF
END_CASE;
WHERE
  SIZEOF(SELF.TYPE) = 1;
END_ENTITY;

```

Abbildung 4.5: Teilemodell, *EXPRESS*-Text

Erwähnenswert an diesem Modell ist, dass sich dieses Konzept auch einfach mit den bereits existierenden Constraintmechanismen kombinieren lässt. So wird

in der lokalen Regel der Entität “Part” sichergestellt, dass jede gültige Instanz genau ein sekundäres Merkmal aufweist.

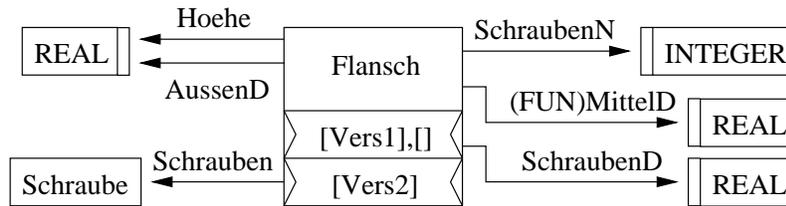


Abbildung 4.6: Flansch, *EXPRESS-G*

Zusätzlich verhindert das Modell den Fall, dass ein Teil durch sich selbst ersetzt wird. Dies geschieht mit einer bedingten, lokalen Regel, unter Verwendung des in *EXPRESS* definierten Vergleichs der Objektbezeichner.

In einem zweiten Beispiel sollen die Auswirkungen einer Schemaevolution dargestellt werden. Gleichzeitig wird auch gezeigt, dass gewisse Fragen der Versionenbehandlung über zusätzliche Klassifizierungsmerkmale gelöst werden können. Letzterer Punkt ist besonders in der Produktdatenindustrie relevant, wenn mehrere Versionen einer Produktbeschreibung gleichzeitig existieren. Die Aenderung betrifft den Schraubendurchmesser in der existierenden Version des Datenschemas. Er soll neu durch eine Assoziation zwischen Flansch und Schraubenbeschreibung festgehalten werden. Damit ist die Information weiterhin vorhanden, auch wenn sie anders strukturiert ist.

Wie die graphische Version des Datenmodells (siehe Abb. 4.6, textuelle Version in Abb. 4.7) zeigt, sind alle Attribute, die nur den Flansch betreffen weiterhin unabhängig von irgendwelchen Sekundärkriterien verwendbar. Der Zugriff auf diese Informationen kann nach den Grundlagen aus Unterkapitel 4.3 effizient erfolgen. Die Instanz muss somit bei der Dereferenzierung nicht auf die Gültigkeit von Sekundärmerkmalen getestet werden.

Um die Instanzen gemäss dem alten Datenmodell weiterhin unterstützen zu können, wurden verschiedene Klassifikationskriterien zur Abgrenzung der verschiedenen Versionen eingeführt. In Abhängigkeit von der Version sind somit die alten oder aber die neuen Definitionen gültig; beide Generationen der Instanzen können gleichzeitig koexistieren. In diesem Fall verlangt der Zugriff auf ein bedingt gültiges Element einen vorgängigen Merkmalstest. Ein Kodefragment, das dies verdeutlicht ist dazu in Abbildung 4.8 zu sehen.

In den Beispielen wird auch deutlich, dass der Aenderungsaufwand bei der Schemaevolution überall dort gross ausfällt, wo die Semantik der Objekte nur in den Anwendungen vorhanden ist. Dieses Problem wird im nächsten Kapitel untersucht.

```

ENTITY Flansch;
  TYPE = SET OF (Vers1, Vers2)
  SchraubenN: INTEGER;
  MittelD:
    FUNCTION(REAL):REAL;
  AussenD: REAL;
  Hoehe: REAL;

  -- state oriented definitions
CASE TYPE OF
  [Vers1], [ ]:
    SchraubenD: REAL;
  [Vers2]:
    Schrauben: Schraube;
END_CASE;
WHERE
  SchraubenN > 0;
  SchraubenN MOD 4 = 0;
  Hoehe > 0.1 * AussenD;
  SELF.TYPE <> [Vers1, Vers2];
END_ENTITY;

```

Abbildung 4.7: Flansch, EXPRESS-Text

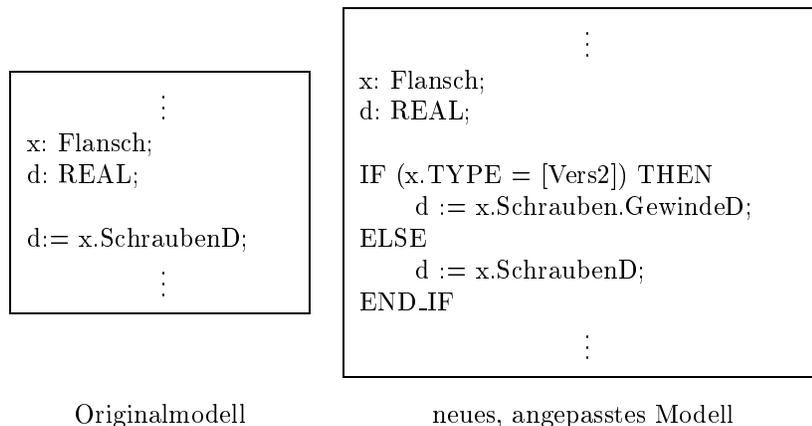


Abbildung 4.8: Kodefragment, Flansch mit Versionen

4.6 Zusammenfassung

In diesem Kapitel wurde ein neuartiger Ansatz zur Erweiterung des Typenkonzepts von *EXPRESS* eingeführt. Dies geschieht mit Hilfe von bedingt gültigen Elementen im objektorientierten Datenmodell.

Nach der Definition der syntaktischen Erweiterung, sind die wichtigsten Grundlagen des neuen Konzepts mit formalen Definitionen untermauert worden. Dabei ergaben sich folgende wesentliche Erkenntnisse:

1. Das Konzept von Sekundärklassifikationen ist konsistent und orthogonal zu dem bereits existierenden Typenkonzept.
2. Typevolution kann unter Verwendung der Grundlagen aus diesem Kapitel eingeführt werden. Dabei resultiert aus dem Zugriff auf Attribute von Instanzen die keiner Evolution unterliegen keine Effizienzeinbusse.
3. Die Schemaevolution wird vereinfacht, indem mit Hilfe von Zuständen die Typproliferation respektive der Anpassungsaufwand an die Modelle reduziert wird.

Zum Schluss ist die Umsetzung des Konzepts auf typische Problemstellungen der Praxis anhand von zwei konkreten Beispielen aufgezeigt worden.

Kapitel 5

Methoden

Wie in Kapitel 2.3.3 dargestellt besteht eine vollständige Definition von Datentypen aus zwei Teilen, einer strukturellen und einer operationellen Spezifikation. Im gleichen Kapitel wurde dargelegt, dass heutige Datenmodelliermittel sich auf die strukturelle Beschreibung beschränken. Als Konsequenz fehlen die semantisch bedeutungsvollen, operationellen Teile. Sie sind einzig in den Anwendungen programmiert und führen so bei jeder Schemaänderung zu Problemen. Daneben zeigt es sich, dass nicht nur die Dokumentation des Datenschemas unvollständig ist, sondern auch Mehrfachimplementationen derselben Funktionalität in verschiedenen Applikationen vorkommen.

Aus diesem Grund soll in diesem Kapitel ein Konzept zur Unterstützung von Methoden erarbeitet werden. Dabei ist zu bedenken, dass der Modellierer der Implementation eine gewisse Freiheit in der Definition von operationellen Elementen haben muss. Dennoch sollen die Schnittstellen zu diesen freieren Teilen im Datenmodell erkennbar sein. Daraus ergibt sich die Notwendigkeit, sowohl die statische Methodendefinitionen für Klassen, wie auch die dynamische Bindung von Methoden an Instanzen unterstützen zu müssen.

5.1 Syntaktische Erweiterungen

Mit der im Kapitel 3 erfolgten Einführung von funktionswertigen Attributen wurde eine gute Grundlage zur Unterstützung von operationellen Eigenschaften in Schemadefinitionen geschaffen. Aus diesem Grund muss die *EXPRESS*-Grammatik nur noch in wenigen Regeln erweitert werden, damit sich Datentypen mit Methoden assoziieren lassen. Die so erweiterten Regeln sind in der nachfolgenden Grammatik wieder mit einem Stern markiert.

```

entity      := 'ENTITY' id [subsuper] ';'
              struct_body
              'END_ENTITY'';'.
struct_body := [dyn_state] {attributes}.
* expl_attr := id {' ,' id} ':' ['OPTIONAL'] basetype ';'
*           | id ':' ['OPTIONAL'] basetype default_val.
* default_val := 'DEFAULT' expression.

```

```

subsuper      := [supertype] [subtype].
subtype       := 'SUBTYPE' 'OF'.
supertype     := 'SUPERTYPE' 'OF' '(' superexpr ')'.
superexpr     := superfact {'AND' | 'ANDOR'} superfact}.
superfact     := id
                | 'ONEOF' '(' superexpr
                    {',' superexpr} ')',
                | '(' superexpr ')'.
where_clause  := 'WHERE' domain_rule ';' {domain_rule ';'}].
domain_rule   := [id ':'] expression.
dyn_state     := 'TYPE' ':'
                'ENUMERATION' 'OF' '(' id {',' id} ')'.
attributes    := {expl_attr}
                ['CASE' 'TYPE' 'OF'
                 {state_set ':'
                  {expl_attr
                   [where_clause]}
                 'END_CASE' ';'}].
state_set     := state {',' state}.
state         := '[' id {',' id} ']'.

* entity_constructor
*           := entity_ref '(' [ attr_val
*                               {',' attr_val}] ')'.
* attr_val   := 'DEFAULT' | expression.

```

Die wichtigste Neuerung findet sich in der erweiterten Regel “expl_attr”. Sie erlaubt die Definition eines Grundwertes für zukünftige Ausprägungen von Entitäten. Indem diese Möglichkeit für einfache wie auch für funktionswertige Attribute definiert ist, konnte eine mühsame Trennung verschiedener Spezialfälle in der Grammatik vermieden werden. Gleichzeitig kann der Datenmodellierer auf diese Art auch Standardwerte für sämtliche Attribute im Datenmodell ausweisen.

Die Existenz eines Initialwertes verlangt natürlich eine Anpassung der Erzeugung von Instanzen. In Datenschemata geschieht dies über Grammatikregel “entity_constructor”. Sie wird dahingehend erweitert, dass statt eines konkreten Wertes das Schlüsselwort “DEFAULT” angegeben werden kann.

Eine Erweiterung der Syntax des Austauschformats nach [ISO94c] erübrigt sich, da mit dem Token ‘*’ bereits ein brauchbares Hilfsmittel für die Bezeichnung ausgelassener Information existiert. Dieses wurde bisher im Spezialfall eines durch ein “derived attribute” redefinierten Attributs verwendet. Somit kann das Zeichen ‘*’ seine bisherige Funktion im gleichen Sinn, als Auslassungszeichen für konkrete Werte in einer erweiterten Verwendung wahrnehmen.

Mit diesen syntaktischen Erweiterungen ist auf einfache Weise eine Art von Methoden eingeführt. Das vorgeschlagene Konzept unterscheidet sich dabei von den üblicherweise verwendeten Methoden darin, dass die Empfängerinstanz des Methodenaufrufes *nicht* als impliziter Parameter übergeben wird (siehe Abbil-

```

FUNCTION dist(pt:GeomPoint, pl:GeomPlane): REAL;
  RETURN ...;
END_FUNCTION;

ENTITY GeomPoint;
DERIVE
  Distance: FUNCTION(GeomPoint, GeomPlane):REAL = dist;
END_ENTITY;

ENTITY GeomPlane;
DERIVE
  Distance: FUNCTION(GeomPoint, GeomPlane):REAL = dist;
END_ENTITY;

```

Abbildung 5.1: Beispiel für Methoden in *EXPRESS*

Abbildung 5.1). Dies erscheint jedoch eher ein Vor- als ein Nachteil zu sein. Bei traditionellen Methoden ist nämlich oft nicht klar, an welches Datenobjekt die Funktionalität gebunden werden soll. So lässt sich eine Funktion zur Berechnung des Abstandes zwischen einem geometrischen Punkt und einer Ebene sowohl an den Datentyp "Ebene" als auch an Punktobjekte binden. Als Abhilfe werden die Methoden dann oft mehrfach implementiert und separat an die beiden Datentypen gebunden. Alternativ (siehe Abb. 5.2) wird die Methode einmal implementiert, an den einen Typ gebunden und vom anderen mit vertauschten Argumenten aufgerufen.

```

//
//show how cross-associations work in C++
//
class GeomPlane; //forward declaration
class GeomPoint
{ //first some attributes then...
public:
  double distance(GeomPlane *arg)
  { return ...;};
};
class GeomPlane
{ //first some attributes then...
public:
  double distance(GeomPoint *arg)
  { return arg->distance(this);};
};

```

Abbildung 5.2: Kreuzassoziation von Methoden in C++

Diese Probleme treten bei der expliziten Uebergabe von Parametern nicht auf, da eine Operation mit mehreren Datentypen assoziiert werden kann.

indem wie bis anhin Attributwerte explizit angegeben werden. Als Alternative kann auf die explizite Nennung eines Wertes verzichtet werden, wenn im Datenschema ein Standardwert definiert ist. Dabei ist durch die zuvor verlangte Evaluierbarkeit des Ausdrucks “ N ” sichergestellt, dass die so verwendeten Werte wohldefiniert sind.

(Val StateSimpleEnt)

$$\Gamma \vdash A_1, \dots, A_m, T_1, \dots, T_p, N_1 : T_1, \dots, N_p : T_p, N : T$$

$$B = ENTITY(l_1 : A_1, \dots, l_j : A_j \downarrow N, \dots, l_m : A_m) \{s_k\} SUB(), x : B$$

$$m, j, k \in 0 \dots, j, p \in 1, \dots, m, T \preceq^* A_j$$

$$s \subseteq \{s_k\} \quad V = \langle A_i, i \in 0 \dots m \mid visible(B, l_i, \{s\}) \rangle$$

$$\forall A_i \in V \bullet T_i \preceq^* A_i$$

$$\Gamma, Ref \ x : B(s, N_1, \dots, N_{j-1}, *, N_{j+1}, \dots, N_m) \vdash \diamond$$

Diese verbale Definition wird durch die Regel “Val StateSimpleEnt” wiedergegeben und in der Regel “Val StateComplEnt” rekursiv erweitert, um auch komplexe Instanzen erzeugen zu können.

(Val StateComplEnt)

$$\Gamma \vdash A_1, \dots, A_m, \{s_j\}_1 B_1, \dots, \{s_j\}_n B_n, N_1 : T_1, \dots, N_p : T_p,$$

$$N : T, B = ENTITY(l_1 : A_1, \dots, l_q : A_q \downarrow N, \dots, l_m : A_m) \{s_k\}$$

$$SUB(B_1, \dots, B_n), x : B$$

$$m, k \in 0 \dots, p, q \in 1, \dots, m, n \in 1 \dots, T \preceq^* A_q$$

$$s \subseteq \{s_k\} \cup \bigcup_{i=1 \dots n} \{s_j\}_i, \quad V = \langle A_i, i \in 0 \dots m \mid visible(B, l_i, \{s\}) \rangle$$

$$\forall A_i \in V \bullet T_i \preceq^* A_i$$

$$\Gamma, Ref \ x : B(s, N_1, \dots, N_{j-1}, *, N_{j+1}, \dots, N_p) \bigoplus_{i=1 \dots n} B_i(\dots) \vdash \diamond$$

Bisher noch nicht angesprochen wurde die Dereferenzierung von funktionswertigen Attributen und der gleichzeitige Aufruf der daraus resultierenden Funktion. Diese Definition gestaltet man zweckmässigerweise als Verkettung von zwei Grundoperationen. Zum einen ist das die Dereferenzierung, wie sie in der Regel “Deref” im Kapitel 4.3 eingeführt wurde. Zusätzlich kommt noch der eigentliche Funktionsaufruf hinzu (siehe “ExecFunc”).

(ExecFunc)

$$\begin{array}{c}
\Gamma \vdash F_1, \dots, F_m, F, A = \mathit{FUNC}(F_1, \dots, F_m) : F \\
N_1 : T_1, \dots, N_m : T_m, f : A \\
m \in 0 \dots, \forall j = 1 \dots m \bullet T_j \preceq^* F_j \\
\hline
\Gamma \vdash f(N_1, \dots, N_m) : F
\end{array}$$

Die Regel ist recht einfach, wenn man von einer Variablen eines Funktionstyps mit der Signatur “ $(F_1, \dots, F_m) : F$ ” ausgeht. Eine mit der Variablen “ f ” referenzierte Funktion kann nun mit Angabe von Argumenten aufgerufen werden.

Die Verknüpfung der Regeln “Deref” und ”ExecFunc” liefert die gesuchte Funktionalität eines Methodenaufrufes. Das geschieht, indem in einem ersten Schritt ein Attribut dereferenziert und danach die resultierende Funktionsreferenz unter Angabe von Werten für die Parameter aufgerufen wird.

Zum Schluss fehlt noch eine Definition zur Unterstützung der Instanzenmigration. Wie im Abschnitt über die Objektevolution (siehe Kapitel 4.4.1) festgehalten, müssen bei der Typerweiterung allenfalls fehlende Attributwerte ergänzt werden. Eine einfache und konsistente Semantik ergibt sich, wenn man die Standardwerte bei einer Zustandsänderung mitberücksichtigt. Dies führt auf die Regel “StateChange” mit der aufgezeigten operationellen Semantik.

(StateChange)

$$\begin{array}{c}
\Gamma \vdash A_i, N : T, \{s_j\}_1 B_1, \dots, \{s_j\}_n B_n, \\
B = \mathit{ENTITY}(\dots, l_i : A_i \downarrow N, \dots) \{s_k\} \mathit{SUB}(B_1, \dots, B_n), \\
x : B, \mathit{Ref} x : B(s, \dots) \\
T \preceq^* A_i, S \subseteq \{s_k\} \cup \bigcup_{i=1 \dots n} \{s_j\}_i, s \neq S, \\
\text{NOT } \mathit{visible}(B, l_i, \{s\}) \wedge \mathit{visible}(B, l_i, \{S\}) \\
\hline
x. \mathit{TYPE} := S \Rightarrow x.l_i = N
\end{array}$$

Das bedeutet also, dass die Standardwerte bei einem Zustandswechsel zur sicheren Initialisierung der neu gültigen Attribute herangezogen werden.

5.3 Ausführlichere Beispiele

Als erstes Beispiel wird die Nachbildung von Methoden, wie sie in existierenden Programmiersprachen eingesetzt werden, verdeutlicht. Im Beispiel sollen Volumeninformationen über den kleinsten umgebenden Quader von geometrischen

```

FUNCTION BBVolGeom(arg: Geometry): REAL;
  RETURN 0.0; -- general object;
END_FUNCTION;

ENTITY Geometry
  ABSTRACT SUPERTYPE OF ONEOF(Flansch, Schraube);
DERIVE
  BoundBoxVol: FUNCTION(Geometry): REAL =
    BBVolGeom;
END_ENTITY;

```

Abbildung 5.3: Teilemodell mit Methoden, *EXPRESS*-Text

Objekten mit in das Datenschema aufgenommen werden. Solche Informationen werden zur Lösung von Transport-, Verpackungs- und Assemblyproblemen verwendet.

Für solche funktionalen Eigenschaften bieten sich Methoden an. Daher wird ausgehend vom bereits früher verwendeten Beispieldatenmodell in einem ersten Schritt eine Methode für das (abstrakte) Wurzelobjekt “Geometry” eingeführt (siehe Abb. 5.3). Dieser Datentyp dient nur als Bezeichner für allgemeine Eigenschaften aller Geometrieklassen. Daher liefert die entsprechende Methode (“BBVolGeom”) vorerst ein konstantes Nullvolumen.

Fortan erbt jede aus dem Typ “Geometry” hervorgehende Klasse die Eigenschaft, das Volumen einer “Bounding Box” berechnen zu können. Um den spezialisierten Datentyp akkurat definieren zu können, muss die Methode allerdings bei Bedarf überschrieben werden, wie das im Beispiel der Entität “Flansch” (siehe Abb. 5.4) gezeigt wird.

Das Kodefragment in Abb. 5.5 zeigt, wie solche Methoden verwendet werden können. Dabei stellt die Wahl eines abgeleiteten Attributs sicher, dass die Methode vom Laufzeitsystem *nicht* auf Instanzenbasis geändert werden kann.

Im zweiten Beispiel wird die Verwendung des Methodenkonzepts für an Instanzen gebundene Operationen vorgestellt. Im Datenmodell werden für solche Methoden einzig die Anknüpfungspunkte und eine Initialisierung definiert. Jede Instanz kann später ihr operationelles Verhalten durch Aenderung der assoziierten Methode an die individuellen Eigenschaften anpassen.

Eine sinnvolle Verwendung für diese Modelliermöglichkeit wird am Beispiel von lokalen Konsistenzregeln gezeigt. Oft kann der Datenmodellierer nur einen Teil der Konsistenzregeln detaillieren. Bisher gibt es keine Möglichkeit in *EXPRESS*, das Fehlen der vollständigen Definition im Schema darzustellen, oder dem Benutzer eine Schnittstelle zur Vervollständigung der Regeln zu Verfügung zu stellen. Gerade letzteres ist aber in Anbetracht der spezifischen Verwendung

```

FUNCTION BBVolFlansch(arg: Flansch): REAL;
  RETURN arg.AussenD*arg.AussenD*arg.Hoehe;
END_FUNCTION;

ENTITY Flansch;

  -- wie in Abb. 4.7 definiert

DERIVE
  SELF\Geometry.BoundingBoxVol: FUNCTION(Flansch):REAL
    = BBVolFlansch;

  -- wie in Abb. 4.7 definiert
  :
END_ENTITY;

```

Abbildung 5.4: Flansch mit Methoden, *EXPRESS*-Text

```

x: Flansch;
volume: REAL;

x := Flansch(...) || Geometry();
volume := x.BoundingBoxVol(x);    - - method call
:

```

Abbildung 5.5: Verwendung von Methoden, *EXPRESS*-Text

```

FUNCTION NotBad(t: Part):LOGICAL;
  RETURN UNKNOWN;
END_FUNCTION;

ENTITY Part;
  TYPE = SET OF(stocked, ordered, replaced);
  Key: LABEL;
  AssocShape: Geometry;
  UserRule := FUNCTION(Part):LOGICAL DEFAULT NotBad;
  -- state oriented definitions
  CASE TYPE OF
    [stocked]:
      QuantityAv: INTEGER;
    [ordered]:
      QuantityOr: INTEGER;
      DateExpDel: Date;
    [replaced]:
      ReplacedBy: Part;
      WHERE ReplacedBy :<>: SELF
  END_CASE;
  WHERE
    R1: (SIZEOF(SELF.TYPE) = 1) OR
        (NOT ({replaced} IN SELF.TYPE));
    R2: SELF.UserRule(SELF)
  END_ENTITY;

```

Abbildung 5.6: dynamisch bindbare Methode

komplexer Daten und den damit verbundenen komplizierten Konsistenzbedingungen ein äusserst brauchbares Hilfsmittel.

In Abbildung 5.6 wird gezeigt, wie das Teilemodell um eine Methode erweitert wird (“UserRule”). Diese soll die nachträgliche Spezifikation von lokalen Konsistenzbedingungen durch den Benutzer erlauben. Im Datenmodell wird für die später definierte Konsistenzfunktion ein sinnvoller Standardwert vorgegeben (“NotBad”), und gleichzeitig wird die bereits existierende lokale Regel so modifiziert, dass auf die dynamisch bindbare Funktion zurückgegriffen werden kann (“R2”). In einem Austauschfile kann die Implementation ihre eigene Version der Konsistenzregel mit der Instanz assoziieren und an eine andere Implementation austauschen (siehe Abb 5.7).

Das Datenschema schreibt in dem Beispiel vor, dass in Instanzen vom Typ “Part” das Sekundärmerkmal “replaced” nicht mit andern Merkmalen kombiniert werden darf (siehe Abb. 5.6, Regel “R1”). Der Anwender legt durch die Verwendung seiner Regel fest, dass in seiner Implementation die Sekundärkriterien {*stocked*, *ordered*} kombiniert werden dürfen, falls weniger als zehn Teile vorrätig sind (Bindung über Regel “R2”, siehe auch Instanz #1 in Abb. 5.7).

```

#1=FUNCTION(a:Part):LOGICAL;
    RETURN (SIZEOF(a.TYPE) = 1) OR (a.QuantityAv<10);
END_FUNCTION;
#2=FUNCTION(x:REAL):REAL;
    :
    END_FUNCTION;
#3=Flansch({Vers1}, 8, #2, 16, 20, 2) Geometry()
#4=Part({stocked}, "0345A35", #2, #1, 100);

```

Abbildung 5.7: Austauschfile mit dyn. Methode

5.4 Zusammenfassung

In diesem Kapitel konnte gezeigt werden, dass die existierende Datemodelliersprache *EXPRESS* mit wenig Aufwand um methodenartige Eigenschaften erweitert werden kann. Aus den operationellen Erweiterungen ergeben sich die folgenden Vorteile in der Datenmodellierung:

1. Datentypen lassen sich auch bezüglich ihrer funktionalen Eigenschaften spezifiziert, indem man Operationen mit Typen assoziieren kann.
2. Das Uberschreiben von Methoden zur Definitionszeit erleichtert den modularen Aufbau von Datenmodellen und die Wiederverwendbarkeit von Datentypen.
3. Die Möglichkeit, Methoden zur Laufzeit zu überschreiben erleichtert die Individualisierung von Ausprägungen durch den Benutzer.
4. Die Verfügbarkeit von Standardwerten unterstützt die Typevolution, indem eine Initialisierung von bedingten Attributen ermöglicht wird.

Kapitel 6

Ausblick

6.1 Erreichte Ziele

In dieser Arbeit zeigt sich deutlich, dass die Datenmodelliersprache *EXPRESS* wie sie als Standard existiert zwar Schwachstellen aufweist, im Kern aber auch heute, über zehn Jahre nach ihrer Konzeption noch brauchbar ist. Trotz des Siegeszugs von objektorientierten Ansätzen und Datenbanken – speziell im Bereich der Produktdatenverwaltung – finden sich bis heute kaum brauchbare Alternativen zur Spezifikation von objektorientierten Datenschemata.

Mehrfach wurde erwähnt, dass heutige Produktdatenmodelle sich durch ihre Komplexität von vielen anderen Anwendungen unterscheiden. Die daraus entstandenen Anforderungen konnte *EXPRESS* nicht vollumfänglich erfüllen. Dennoch spricht es für die Sprachendesigner, dass sie nicht nur viele Entwicklungen vorweggenommen, sondern der Erweiterung von *EXPRESS* auch kaum Hürden in den Weg gestellt haben.

Mit relativ wenig Aufwand konnten deshalb parametrische Daten sowohl im Datenmodell als auch im Austauschformat unterstützt werden. Die hier gemachten Ansätze vermögen mittelfristig auftauchende Anforderungen zu lösen. Durch die eingeführten Erweiterungen, die übrigens beinahe vollständig aufwärtskompatibel sind und damit keine existierenden Datenmodelle invalidieren, konnte dem Aspekt der dynamischen Entwicklung von Modellen und Instanzen Rechnung getragen werden. Traditionell werden Typeevolutionen von Instanzen in Programmsystemen kaum und in Datenmodelliersprachen überhaupt nicht unterstützt. Ein Mangel der sich bei den heutigen rechtlichen, marktwirtschaftlichen und informatiktechnischen Anforderungen durch Probleme bei der Datenspeicherung und -archivierung negativ auswirkt. Mit der hier vorgestellten Integration solcher Eigenschaften in das Datenmodell gewinnt man nicht nur einen Mehrwert in der Dokumentation des Modells, sondern auch einen langfristig stabilen Datenbestand.

Schlussendlich ist auch der Gewinn durch den Einsatz von formalen Mitteln zur Beschreibung der Semantik von Konstrukten zu erwähnen. Die Wahl fiel auf eine Notation, wie sie in der Typtheorie in der einen oder anderen Variante geläufig ist [CA96]. Als Alternative hätte man sicher auch die Spezifikationsprache Z [WD96, Inc92, PST91] wählen können. Die Wahl des Notationsmit-

tels spielt eigentlich nur eine untergeordnete Rolle, wichtiger ist die Erkenntnis, dass

1. sich die Semantik beliebig genau beschreiben lässt, jedoch immer ein undefinierter Rest bleibt, der implizite Annahmen oder allgemeines Verständnis voraussetzt.
2. die Wahl einer formalen Sprache gerechtfertigt ist. Diese Form der Definition fördert ein systematisches Durchdenken der Vorbedingungen und Konsequenzen, erleichtert das Erkennen von Widersprüchen und erlaubt ein einfacheres Ableiten einer Implementierung.

6.2 Grenzen der Erweiterungen

Ein grundsätzliches Problem von *EXPRESS* liegt im Fehlen einer formalen semantischen Grundlage. So basieren die im Rahmen von STEP erarbeiteten Modelle mehrheitlich auf einem gemeinsamen Verständnis der Modellersprache, doch werden sich die daraus entstandenen Definitionen erst ausserhalb der Entwicklergemeinschaft bewähren müssen.

Die in dieser Arbeit vorgestellten Erweiterungen sind in sich abgeschlossen. Somit sollten sie nach gegenwärtiger Interpretation der Semantik von *EXPRESS* zu keinen Problemen mit bestehenden Datenmodellen führen. Diese Annahme ist jedoch mangels einer formalen Basis von *EXPRESS* nicht beweisbar. Ebenfalls wäre es aus demselben Grund nicht möglich, allfällig auftretende Inkompatibilitäten im Detail zu ergründen.

Damit kommt man bereits zu weiteren Grenzen, die in *EXPRESS* selbst, aber auch in jeder Erweiterung zu finden sind. So weist die Datenmodellersprache einen sehr hohen Grad an Komplexität auf. Dies mag für gewisse Anwendungen notwendig sein, um komplizierte Probleme in einem Datenschema modellieren zu können. In anderen Bereichen sind hingegen zu viele Konzessionen an die Benutzerwünsche gemacht worden. Das führt zu einer Aufnahme von "Gefälligkeitskonstrukten", die eigentlich nicht notwendig sind. Jede Erweiterung trägt ihren Teil dazu bei, diese Implementationen, wie auch das Verständnis der Modelliermöglichkeiten weiter zu erschweren. Eine weitere Konsequenz daraus ist, dass viele Konzepte nicht oder nur beschränkt in der Praxis erprobt werden konnten.

Auf technischer Ebene stellen sich den Erweiterungen zwei sehr direkte Grenzen. Es ist nicht möglich, **asynchrone Ereignisse** beschreiben zu können. Falls die Funktionsweise von Produkten spezifiziert werden soll, so wäre eine eigentliches Laufzeitmodell mit den Möglichkeiten zur Spezifikation von asynchronen Ereignissen notwendig. Spätestens dann ist jedoch die Trennung zwischen Implementations- und Spezifikationssprache in der Praxis nicht mehr erkennbar. *EXPRESS* hätte in einem solchen Umfeld derart weitgehende Definitionen, dass eine Implementation direkt in einem *EXPRESS*-System fast zwingend erschiene.

Die zweite praktische Grenze liegt darin, dass die Datenschemata immer noch eine statische Definition des Erlaubten darstellen. Die Erweiterungen aus der vorliegenden Arbeit vermögen diese Tatsache etwas aufzuweichen. Eine vollkommen **dynamische Erweiterung** der Attribute oder Funktionalität von Datentypen, sowie der Weitergabe dieser Information an einen Datenaustauschpartner ist immer noch nicht möglich. Aus Sicht der Standardisierung ist das hingegen kein Problem, sondern eher eine gewünschte Eigenschaft. In der allgemeinen Datenmodellierung könnten Datenmodelliersprachen jedoch bedeutend leistungsfähiger werden, wenn sich die Modellinformationen dynamisch anpassen lassen und die Daten selbstdokumentierend werden.

6.3 Zukünftige Schritte

Der wichtigste nächste Schritt besteht darin, die vorgeschlagenen Erweiterungen zu implementieren. Dabei kann getestet werden, wie weit sich die Aussagen bezüglich Laufzeiteigenschaften und Widerspruchsfreiheit tatsächlich bewahren.

Vor einer weiteren Erweiterung der Modelliereigenschaften ist eine solide, formale Semantik der gesamten Sprache unabdingbar. Nur so lässt sich die gute Grundlage langfristig verwenden und weiterentwickeln.

Dringend notwendig ist ebenfalls eine Abklärung, was in Zukunft angestrebt werden soll. Bis anhin sind Eigenschaften in *EXPRESS* integriert worden, die aus dem Ziel der Modellierung von Produktdaten stammten. Dabei wurde vorwiegend eine strukturelle Beschreibung erwartet. Mit dem Einsatz der Sprache zu andern Zwecken, ändert sich auch das Anforderungsprofil. Das wiederum führt zu teilweise in Widerspruch stehenden Forderungen. Die Erfahrung hat jedoch gezeigt, dass grosse und komplexe Systeme gegenüber den kleinen, wohlgedachten Spezialhilfsmitteln unterlegen sind. Auch dies ist ein weiterer Grund, weshalb man zuerst die Mängel der existierenden Modelliersprache anhand Implementierungen untersuchen sollte.

Glossar

ANSI American National Standards Institute

AP, Application Protocol als Anwendungsprotokoll wird eine Form von Datenmodellen in STEP bezeichnet. Im Unterschied zu anderen Modellen repräsentieren Anwendungsprotokolle das branchen- und anwendungsspezifische Wissen.

ASCII Abkürzung für *American National Standard Code for Information Interchange* [ASC86]. Die am weitesten verbreitete Kodierungen für druckbare Zeichen sowie Steuerbefehle.

CAD *Computer Aided Design* ein Computerprogramm, das einem Entwickler hauptsächlich beim Entwurf von Plänen für geometrische Eigenschaften von Produkten hilft.

CORBA – bei der *Common Object Request Broker Architecture* handelt es sich um eine Definition von Schichten und Protokollen zur Verwendung von Objekten in inhomogenen Computernetzen. Objekte können verteilt gespeichert und von verschiedenen Systemen und Applikationen verwendet werden.

Datentyp Ein abstraktes Gebilde zur Klassifikation von Realweltobjekten. Eine Datentypdefinition besteht aus einem strukturellen und einem operationellen Teil und dient der sicheren Datenmanipulation in Computersystemen.

Entität Mit Entität werden in *EXPRESS* Datentypen bezeichnet, die Eigenschaften von Klassen in objektorientierten Sprachen aufweisen.

EXPRESS standardisierte, formale Sprache [ISO94b] zur Definition von Datenschemata.

first class objects wird meistens als Attribut zu Funktionen verwendet. Damit soll angedeutet werden, dass sämtliche Operationen, die auf gewöhnliche Objekte anwendbar sind auch auf Funktionen angewendet werden dürfen. Typischerweise beinhaltet das die Möglichkeit, Funktionen einer Variablen zuzuweisen, oder Funktionen abzuspeichern.

Guard, Typeguard Ein ursprünglich von C. A. R. Hoare in allgemeiner Form eingeführtes Konstrukt zur Sicherstellung von Vorbedingungen in Computersprachen. Dieser Vorschlag wurde später von Niklaus Wirth für Typzusicherungen in Oberon-2 eingesetzt.

- IGES** *Initial Graphics Exchange Specification* ist ein Datenaustauschstandard für Geometrieinformationen, der in der Industrie noch oft eingesetzt wird.
- Instanz** Unter Instanzen werden in *EXPRESS* Ausprägungen von Entitäten verstanden. Sie stellen also die konkretisierten Daten dar.
- ISO** Akronym für *International Standardization Organization*, eine der weltweiten Standardisierungsorganisationen mit Sitz in Genf.
- neutrales File** Eine Sammlung von Daten nach [ISO94c]. Mit “neutral” bezieht man sich auf die Kodierung und die strukturelle Organisation der Daten. Beide sind in einem nichtproprietären, öffentlich verfügbaren Format.
- Objekt** Objekte sind den Datentypen nahestehend. Im Gegensatz zu diesen kann darunter auch ein Individuum der Realwelt verstanden werden. In der Informatikliteratur existiert bis heute keine allgemein akzeptierte Definition für diesen Begriff.
- ODM** *Object Management Group*, eine nicht gewinnorientierte Vereinigung mit etwa 600 Mitgliedern aus den Bereichen Softwareentwicklung, -verkauf und Endbenutzern. Ziel dieser Gruppierung ist die Verbreitung von objektorientierten Technologien. Ein Resultat ihrer Arbeit findet sich in der Object Management Architecture (OMA) sowie der Common Object Request Broker Architecture (CORBA).
- ODMG** Object Database Management Group, die Gruppe, die den Standard ODS entwickelt hat. ODMG und auch ODMG-93 werden synonym für den Standard selbst verwendet. Ziel der Gruppierung ist eine Interoperabilität zwischen objektorientierten Systemen auf Basis des Quelltextes. (Weitere Informationen unter <http://www.odmg.org>)
- ODS** *Object Database Standard* [C⁺94, Cat95] auch bekannt unter dem Namen ODMG-93, ein Standard für objektorientierte Datenbanken, herausgegeben durch ODMG.
- OMA** Bei der *Object Management Architecture* handelt es sich um eine Referenzmodell anhand dessen die Entwicklung und Zusammenarbeit von objektorientierten System vereinfacht werden soll. Die Architektur wurde von ODMG entwickelt.
- Polymorphismus** bezeichnet die Eigenschaft, dass in einer Variable nicht nur ein Objekt vom deklarierten Typ, sondern auch ein daraus abgeleitetes Objekt gespeichert werden kann.
- SET** *Standard d'Echange et de Transfert*, ein im Jahre 1984 entstandener Datenaustauschstandard [SET89], der vorwiegend in Frankreich eingesetzt wird.
- STEP** Abkürzung für *STandard for the Exchange of Product model data*, ein Versuch den Datenaustausch zwischen CAD/CAM Systemen zu vereinfachen. Der Austausch soll unabhängig von Hersteller und Computerarchitektur geschehen. Als Nachfolger von IGES und VDAFS erlaubt der STEP Standard aber eine weitergehende Beschreibung als nur die Geometrie von Produkten.

STEP-File Ist ein anderer Begriff für *neutrales File*.

VDA, VDA-IS, VDA-FS *Verband der Automobilindustrie e.V.*, ein Verband von deutschen Automobilherstellern unter dessen Federführung die Normen VDA-IS (IGES Subset [VDA89]) und VDA-FS (Flächenschnittstelle [VDA86, VDA87]) entwickelt und herausgegeben wurden.

Literaturverzeichnis

- [AA94] Y. Ait-Ameur. Part library: Proposal for list expressions. Technical report, ISO 15384 TC184/SC4/WG2, 1994.
- [ABC⁺83] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. An Approach to Persistent Programming. *The Computer Journal*, 26(4):360–365, 1983. Also published in *Readings in Object-Oriented Database Systems*, Stanley Zdonik and David Maier, eds., Morgan Kaufman, San Mateo, California 1990.
- [ABC⁺94] O. Agesen, L. Bak, C. Chambers, B. Chang, U. Hölzle, J. Maloney, R.B. Smith, D. Ungar, and M. Wolczko. *How to use Self 3.0 & The Self 3.0 Programmer's Reference Manual*, 1994.
- [AC96] Martin Abadi and Luca Cardelli. An interpretation of objects and object types. In *POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 396–409. Association for Computing Machinery, January 1996.
- [AS93] H. Abelson and G.J. Sussman. *Structure and Interpretation of Computer Programs*. Springer, 1993.
- [ASC86] American National Standard Code for Information Interchange (ASCII), 1986. ANSI X3.4-1986 (R 1992), revised in 1992.
- [BCM88] Roland Backhouse, Paul Chrisholm, and Grant Malcolm. Do-it-yourself Type Theory. Technical report, Department of Mathematics and Computer Science of Groningen University, 1988.
- [BKKK87] J. Banerjee, W. Kim, H.J. Kim, and H.F. Korth. Semantics and Implementation of Schema Evolution in Object Oriented Databases. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, San Francisco, CA, May 1987.
- [BMO⁺89] Robert Bretl, David Maier, Allen Otis, Jason Penney, Bruce Schuchardt, Jacob Stein, E. Harold Williams, and Monty Williams. *The GemStone Data Management System*. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, chapter 12, pages 283–308. Addison Wesley Publishing Company, 1989.
- [C⁺94] R.G.G. Cattell et al., editors. *The Object Database Standard: ODMG-93*. Morgan Kaufmann Publishers, San Mateo CA, 1994.

- [CA96] L. Cardelli and M. Ambadi. *A Theory of Objects*. Springer Verlag, September 1996. ISBN 0-387-94775-2.
- [Car85] L. Cardelli. Amber. Technical report, AT&T, Bell Labs, Murray Hill, USA, 1985.
- [Cat95] R.G.G. Cattell. Object models and standards. In C. Goble and J. Keane, editors, *Advances in Databases*, number 940 in Lecture Notes in Computer Science. SunSoft Inc, CA, USA, July 1995.
- [Cha93] Craig Chambers. Predicate classes. In Oscar Nierstrasz, editor, *ECOOP 1993 – Object Oriented Programming, 7th European Conference*, volume 707, pages 268–296, July 1993.
- [Che76] P.P. Chen. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, January 1976.
- [CLR93] Colin Charlton, Paul Leng, and Mark Rivers. An Object-Oriented Model of Design Evolution. Technical report, University of Liverpool, 1993.
- [Coc90] W. Paul Cockshott. *PS-ALGOL IMPLEMENTATIONS, Applications in Persistent Object-Oriented Programming*. Ellis Horwood, 1990.
- [CY91] Peter Coad and Edward Young. *OOD, Object Oriented Design*. Prentice Hall, 1991.
- [CY94] Peter Coad und Edward Young. *OOA, Objektorientierte Analyse*. Prentice Hall Verlag, 1994.
- [Dav87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA '87, Object Oriented Programming Systems, Languages and Applications*, pages 227–241. ACM Press, 1987.
- [DD93] C.J. Date and Hugh Darwen. *A Guide to the SQL Standard*. Addison Wesley, 3rd edition, 1993. Covers “SQL2”.
- [EH74] M. Engeli und V. Hrdliczka. *Euklid — Eine Einführung*. Fides Rechenzentrum, 1974.
- [ELP88] E. Engeler, P. Läuchle, und R. Peikert. *Berechnungstheorie für Informatiker*. B.G. Teubner, Stuttgart, 1988.
- [Eng69] M.E. Engeli. Formula Manipulation — The User’s Point of View. In Julius T. Tou, editor, *Advances in Information Systems Science*, volume 1, pages 117–171. 1969.
- [Fra94] Michael Franz. Protocol Extension: A Technique for Structuring Large Extensible Software-Systems. Technical report, Institute for Computer Systems, ETH Zürich, 1994.
- [Ghe90] G. Ghelli. A Class Abstraction for a Hierarchical Type System. *Proceedings of the 2nd Intl. Conf. on Database Theory, ICDT'90*, pages 56–70, 1990.

- [GR83] A. Goldberg and D. Robson. *Smalltalk 80: The Language and its Implementation*. Addison Wesley, 1983.
- [Gra91] Grady Booch. *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., 1991.
- [GSR96] Georg Gottlob, Michael Schrefl, and Brigitte Röck. Extending Object-Oriented Systems with Roles. *Transactions of Database Systems*, 4, July 1996.
- [Har96] E.R. Harold. *Brewing Java: A Tutorial*, 1996. Available at <http://www.eng.auburn.edu/users/rayh/java/javatutorial.html>.
- [Hed90] Raymund Hedirich. *Ein Beitrag zur Konzeption und Anwendung parametrisierter, integrierter Produktmodelle in CAD-Systemen*. Dissertation, Universität (TH) Karlsruhe, 1990.
- [IGE87] Initial Graphics Exchange Specification (IGES) – Digital Representation for Communication of Product Definition Data, 1987. ANSI Y14.226M.
- [IGE91] IGES Recommended Practices Committee, IGES/PDES Organization. IGES 5.0 recommended practices guide, 1991. NISTIR 4600.
- [Inc92] D.C. Ince. *An Introduction to Discrete Mathematics, Formal System Specification and Z*. Oxford Applied Mathematics and Computing Science Series. Clarendon Press, Oxford, 1992.
- [ISO94a] Geneva ISO. ISO 10303, Industrial automation systems and integration — Product data representation and exchange, Part 1: Overview and Fundamental Principles, 1994. *Available through national standards bodies as ISO 10303-1:1994E*.
- [ISO94b] Geneva ISO. ISO 10303, Industrial automation systems and integration — Product data representation and exchange, Part 11: The EXPRESS Language Reference Manual, 1994. *Available through national standards bodies as ISO 10303-11:1994E*.
- [ISO94c] Geneva ISO. ISO 10303, Industrial automation systems and integration — Product data representation and exchange, Part 21: Clear Text Encoding of the Exchange Structure, 1994. *Available through national standards bodies as ISO 10303-21:1994E*.
- [ISO94d] Geneva ISO. ISO 10303, Industrial automation systems and integration — Product data representation and exchange, Part 41: Integrated generic resources: Fundamentals of Product Description and Support, 1994. *Available through national standards bodies as ISO 10303-41:1994E*.
- [ISO94e] Geneva ISO. ISO 10303, Industrial automation systems and integration — Product data representation and exchange, Part 44: Integrated generic resources: Product Structure Configuration, 1994. *Available through national standards bodies as ISO 10303-44:1994E*.

- [ISO94f] Geneva ISO. ISO 10303, Industrial automation systems and integration — Product data representation and exchange, Part 203: Application protocol: Configuration Controlled Design, 1994. *Available through national standards bodies as ISO 10303-203:1994E.*
- [KBC⁺89] Won Kim, Nat Ballou, Hong-Tai Chou, Jorge F. Garza, and Darrel Woelk. *Features of the ORION Object-Oriented Database System.* In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, chapter 11, pages 251–282. Addison Wesley Publishing Company, 1989.
- [Kee89] S.E. Keene. *Object Oriented Programming in Common Lisp.* Addison Wesley, 1989.
- [KGK⁺95] William Kelley, Sunit Gala, Won Kim, Tom Reyes, and Bruce Graham. Schema Architecture of the UniSQL/M Multidatabase System. In Won Kim, editor, *Modern Database Systems*, chapter 30, pages 621–648. Addison Wesley, 1995.
- [KS86] Henry F. Korth and Abraham Silberschatz. *Database Systems Concepts.* McGraw Hill Book Company, 1986.
- [Löf84] Martin Löf. Constructive mathematics and computer programming. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Computer Programming*, pages 167–184. Prentice Hall, 1984.
- [LS93] Peter Lüthy und Marc Schenk. Beurteilung der parametrischen Möglichkeiten von CAD-/CAM-Systemen. Semesterarbeit, Eidgenössische Technische Hochschule, ETHZ, August 1993.
- [MABD88] R. Morrison, M.P. Atkinson, A.L. Brown, and A. Dearle. Bindings in Persistent Programming Languages. *ACM SIGPLAN Notices*, 23(4):27–34, 1988.
- [Mas78] VSM Verein Schweizerischer Maschinenindustrieller. *Normenauszug für Technische Schulen.* VSM Normenbüro, 1978.
- [Mat93] Florian Matthes. *Persistente Objektsysteme.* Springer Verlag, 1993.
- [MBCD89] R. Morrison, A.L. Brown, R. Connor, and A. Dearle. The Napier88 Reference Manual. Persistent programming research report, University of Glasgow & St. Andrews, 1989.
- [Mös94] H.-P. Mössenböck. *Objektorientierte Programmierung in Oberon-2.* Springer-Verlag, Berlin, 1994.
- [NSWW96] M. C. Norrie, A. Steiner, A. Würgler, and M. Wunderli. A Model for Classification Structures with Evolution Control. In *Proceedings of the 15th Conference on Conceptual Modelling ER'96, Cottbus, Germany*, October 1996.
- [Ode89] Martin Odersky. *A New Approach to Formal Language Definition and its Application to Oberon.* PhD thesis, Swiss Federal Inst. of Technology, ETH Zentrum, 8092 Zürich, Switzerland, 1989.

- [Os88] Sylvia L. Osborn. The Role of Polymorphism in Schema Evolution in an Object-Oriented Database. Technical report, Univ. of Western Ontario, London Ontario, Canada N6A-5B7, 1988.
- [Owe93] Jon Owen. *STEP, An Introduction*. Product Data Engineering, Information Geometers Ltd, 1993.
- [Pae93] Andreas Paepcke, editor. *Object-Oriented Programming, The CLOS Perspective*. The MIT Press, Cambridge, Massachusetts, 1993.
- [PÖ95] Randal J. Peters and M. Tamer Özsu. Axiomatization of Dynamic Schema Evolution in Objectbases. In *Proceedings of the 11th International Conference on Data Engineering (ICDE'95)*. IEEE, March 1995.
- [Pri94] David M. Price. *10303-21, Product Data Representation and Exchange, Standard Data Access Interface Specification (SDAI)*. ISO, Geneva, 1994. ISO Balloting Paper, 1.11.93.
- [PS87] J. Penney and J. Stein. Class Modification in the GemStone Object-Oriented DBMS. In *Proceedings of the 2nd International Conference on Object-Oriented Programming Systems*, Orlando, FL, October 1987.
- [PST91] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z*. Series in Computer Science. Prentice Hall, 1991.
- [RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.
- [Rod93] J.F. Roddick. Implementing Schema Evolution in Relational Database Systems: An Approach Based on Historical Schemata. Technical report, Department of Computer Science and Computer Engineering, La Trobe University, October 1993.
- [RW92] Martin Reiser und Niklaus Wirth. *Programmieren in Oberon*. ACM Press, NY, 1992.
- [SET89] Industrial automation – external representation of product definition data – data exchange and transfer standard specification version 89-06, 1989. L’association française de normalisation (AFNOR).
- [SK92] A. Sheth and V. Kashyap. So Far (Schematically) yet So Near (Semantically). In D.K. Hsiao, E.J. Neuhold, and R. Sacks-Davis, editors, *Interoperable Database Systems*, number A-25 in IFIP Transactions, pages 283–312. IFIP, 1992.
- [SLR+92] M.H. Scholl, C. Laasch, C. Rich, H.J. Schek, and M. Tresch. the COCOON Object Model. Technical report, Department of Computer Science, ETH Zurich, 1992.
- [Smi91] B. Smith, editor. *Initial Graphics Exchange Specification (IGES), Version 5.1*. IGES/PDES Organization, 1991.

- [SNS94] Günter Staub, Augusto Nieva, and Frank Schönefeld. PISA Information Modelling Language: EXPRESS-C. Technical report, TC184/SC4/WG5(P2), 1994. superseding N70.
- [SS77] J.M. Smith and D.C.P. Smith. Database Abstractions: Aggregation and Specialization. *ACM Transactions on Database Systems*, 2(2):105–133, March 1977.
- [SW86] Brad Smith and Joan Wellington, editors. *Initial Graphics Exchange Specification (IGES), Version 3.0*. Society of Automotive Engineers, Inc., April 1986.
- [SW94] Mc Douglas Schenck and Dr. P. Wilson. *Information Modeling the EXPRESS Way*. Oxford University Press, NY, 1994. ISBN 0-19-508714-3.
- [Tai96] Joachim Taiber. *Entwurf eines feature- und constraint-basierten CAD/CAM-Systems zur fertigungstechnischen Optimierung*. Dissertation, Institut für Werkzeugmaschinen und Fertigung, Eidgenössische Technische Hochschule, ETH Zürich, 1996.
- [VDA86] VDA-FS, “DIN 66301”, 1986. Deutsches Institut für Normung.
- [VDA87] VDA Surface Interface Version 2.0, 1987. Verband der Automobilindustrie, VDA.
- [VDA89] VDA IGES Subset Version 2.0, 1989. Verband der Automobilindustrie, VDA.
- [VH91] Vincent Ventrone and Sandra Heiler. Semantic Heterogeneity as a Result of Domain Evolution. *SIGMOD RECORD*, 20(4):16–20, December 1991.
- [WD96] Jim Woodcock and Jim Davies. *Using Z, Specification, Refinement and Proof*. Series in Computer Science. Prentice Hall, 1996.
- [Wei86] Jerry Weiss. STEP, functional requirements. Technical report, ISO TC184/SC4 N30, 1986. second version as of May 1988.
- [WG92] Niklaus Wirth and Jürg Gutknecht. *Project Oberon, the design of an operating system and compiler*. ACM Press Books, 1992.
- [Wie94] Hans Ulrich Wiedmer. Parts Library: General Resources. Technical report, ISO 15384 TC184/SC4/WG2, 1994.
- [Wir85] Niklaus Wirth. *Programmieren in Modula-2*. Springer-Verlag, 1985.
- [Wür95] A.P. Würigler. Object Model System: An Object Database Management System for the OM Data Model. Master’s thesis, Swiss Federal Institute of Technology, ETHZ, August 1995. Institute for Information Systems.

Lebenslauf

Name: Thomas Bühlmann
Geburtstag: 18. Juli 1967
Nationalität: Schweiz
Bürgerort: Basel-Stadt (BS) und Schlierbach (LU)

- 1974-1978 Gotthelf-Primarschule in Basel-Stadt
- 1978-1980 Progymnasiums (Holbeinschulhaus) in Basel-Stadt
- 1980-1986 Gymnasiums am Kohlenberg (Typus B) in Basel-Stadt
- 1986 Gastaufenthalt an der Columbia University in New York
- 1986 Rekrutenschule, Aarau
- 1987-1992 Studium der Informatik an der Eidgenössischen Technischen Hochschule (ETH) in Zürich mit Vertiefung in den Gebieten der theoretischen Informatik und Systemsoftware. Im Nebenfach Studium der Betriebswirtschaftslehre. Informatikdiplomarbeit unter Prof. Dr. N. Wirth mit einer Arbeit zur Optimierung von Assemblercode der Mac-Version des Oberon Compilers.
- 1992-1996 Wissenschaftlicher Mitarbeiter der ETH am Institut für Werkzeugmaschinen und Fertigung, Betreuung von Vorlesungen und Studentearbeiten sowie Mitarbeit in der Forschung, speziell bei den technischen Grundlagen für die Schaffung des ISO Standards "10303-11, The EXPRESS Language Reference Manual". Daraus entstand die vorliegende Dissertation unter der Aufsicht von Prof. Dr. M. Engeli.